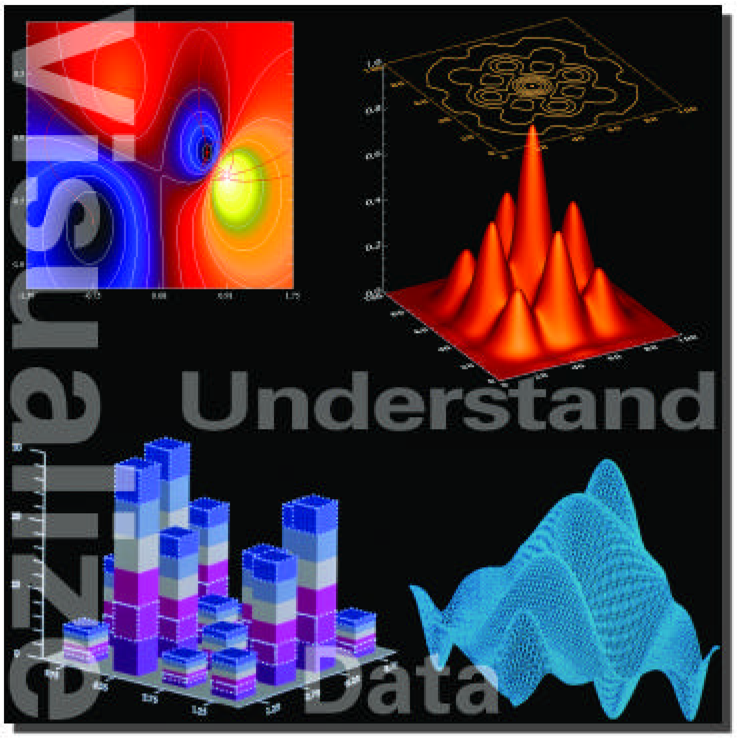
PV-WAVE 7.5®



Signal Processing Toolkit User's Guide

HELPING CUSTOMERS SOLVE COMPLEX PROBLEMS

Visual Numerics, Inc.

Visual Numerics, Inc. 2500 Wilcrest Drive Suite 200 Houston, Texas 77042-2579 United States of America 713-784-3131 800-222-4675 (FAX) 713-781-9260 http://www.vni.com e-mail: info@boulder.vni.com	Visual Numerics, Inc. (France) S.A.R.L. Tour Europe 33 place des Corolles Cedex 07 92049 PARIS LA DEFENSE FRANCE +33-1-46-93-94-20 (FAX) +33-1-46-93-94-39 e-mail: info@vni-paris.fr	Visual Numerics International, Ltd. Suite 1 Centennial Court East Hampstead Road Bracknell, Berkshire RG 12 1 YQ UNITED KINGDOM +01-344-458-700 (FAX) +01-344-458-748 e-mail: info@vniuk.co.uk
Visual Numerics, Inc. 7/F, #510, Sect. 5 Chung Hsiao E. Rd. Taipei, Taiwan 110 ROC +886-2-727-2255 (FAX) +886-2-727-6798 e-mail: info@vni.com.tw	Visual Numerics International GmbH Zettachring 10 D-70567 Stuttgart GERMANY +49-711-13287-0 (FAX) +49-711-13287-99 e-mail: info@visual-numerics.de	Visual Numerics Japan, Inc. Gobancho Hikari Building, 4th Floor 14 Gobancho Chiyoda-Ku, Tokyo, 102 JAPAN +81-3-5211-7760 (FAX) +81-3-5211-7769 e-mail: vda-sprt@vnij.co.jp
VIsual Numerics S.A. de C.V. Cerrada de Berna 3, Tercer Piso Col. Juarez Mexico, D.F. C.P. 06600 Mexico	Visual Numerics, Inc., Korea Rm. 801, Hanshin Bldg. 136-1, Mapo-dong, Mapo-gu Seoul 121-050 Korea	

Table of Contents

Preface

This guide explains how to use the PV-WAVE:Signal Processing Toolkit. The Signal Processing Toolkit provides a broad selection of pre-defined digital signal processing functions, as well as functions that you can customize. It includes both basic and advanced signal processing functions, as well as utility and source code to help you develop your own custom functions.

This manual contains the following parts:

- **Preface** — Describes the contents of this manual, describes the intended audience, lists the typographical conventions used, and explains how to obtain customer support.
- **Chapter 1, *Getting Started*** — Provides a basic overview of the Signal Processing Toolkit along with information and examples to get you started.
- **Chapter 2, *Reference*** — An alphabetically arranged, detailed reference describing each of the functions and procedures in the Signal Processing Toolkit.
- **Appendix A, *Bibliography*** — A complete bibliography of technical literature cited in this manual.
- **Appendix B, *Related Routines*** — A list of PV-WAVE Advantage routines that are useful in digital signal processing.

Intended Audience

This manual is intended for the knowledgeable digital signal processor. No attempt is made to explain basic digital signal processing concepts and techniques.

It is assumed that you are already familiar with PV-WAVE Command Language and/or PV-WAVE Advantage.

Typographical Conventions

The following typographical conventions are used in this guide:

- Code examples appear `in this typeface`.
- Code comments are shown in this typeface.
- Variables are shown in lowercase italics (*myvar*).
- Function and procedure names are shown in all capitals (XYOUTS).
- Keywords are shown in mixed case italic (*XTitle*).
- System variables are shown in regular mixed case type (!Version).

Customer Support

If you have problems installing, unlocking, or running your software, contact Visual Numerics Technical Support by calling:

Office Location	**Phone Number**
Corporate Headquarters Houston, Texas	713-784-3131
Boulder, Colorado	303-939-8920
France	+33-1-46-93-94-20
Germany	+49-711-13287-0
Japan	+81-3-5211-7760
Korea	+82-2-3273-2633
Mexico	+52-5-514-9730
Taiwan	+886-2-727-2255
United Kingdom	+44-1-344-458-700

Users outside the U.S., France, Germany, Japan, Korea, Mexico, Taiwan, and the U.K. can contact their local agents.

Please be prepared to provide the following information when you call for consultation during Visual Numerics business hours:

- Your license number, a six-digit number that can be found on the packing slip accompanying this order. (If you are evaluating the software, just mention that you are from an evaluation site.)
- The name and version number of the product. For example, PV-WAVE 7.0.
- The type of system on which the software is being run. For example, SPARCstation, IBM RS/6000, HP 9000 Series 700.
- The operating system and version number. For example, HP-UX 10.2 or IRIX 6.5.
- A detailed description of the problem.

FAX and E-mail Inquiries

Contact Visual Numerics Technical Support staff by sending a FAX to:

Office Location	FAX Number
Corporate Headquarters	713-781-9260
Boulder, Colorado	303-245-5301
France	+33-1-46-93-94-39
Germany	+49-711-13287-99
Japan	+81-3-5211-7769
Korea	+82-2-3273-2634
Mexico	+52-5-514-4873
Taiwan	+886-2-727-6798
United Kingdom	+44-1-344-458-748

or by sending E-mail to:

Office Location	E-mail Address
Boulder, Colorado	`support@boulder.vni.com`
France	`support@vni-paris.fr`
Germany	`support@visual-numerics.de`
Japan	`vda-sprt@vnij.co.jp`
Korea	`support@vni.co.kr`
Taiwan	`support@vni.com.tw`
United Kingdom	`support@vniuk.co.uk`

Electronic Services

Service	Address
General e-mail	`info@boulder.vni.com`
Support e-mail	`support@boulder.vni.com`
World Wide Web	`http://www.vni.com`
Anonymous FTP	`ftp.boulder.vni.com`
FTP Using URL	`ftp://ftp.boulder.vni.com/VNI/`
PV-WAVE Mailing List:	`Majordomo@boulder.vni.com`
To subscribe include:	`subscribe pv-wave YourEmailAddress`
To post messages	`pv-wave@boulder.vni.com`

Getting Started

Signal processing is widely used in engineering, and scientific research and development for representing, transforming, and manipulating signals and the information they contain. This rapidly advancing technology has applications in many areas including speech processing, data communications, acoustics, radar, sonar, seismology, remote sensing, scientific and medical instrumentation, consumer electronics, time-series analysis, and finance.

The PV-WAVE:Signal Processing Toolkit is a collection of digital signal processing (DSP) functions that work in conjunction with PV-WAVE. This chapter discusses the following main categories of functions found in the Signal Processing Toolkit:

- Signals and Systems (Models and Analysis)
- Filter Approximation
- Filter Realization
- Transforms and Spectrum Analysis
- Statistical Signal Processing
- Polynomial Manipulation
- Optimization
- Plotting and Signal Generation

The Signal Processing Toolkit functions are designed for easy use, while providing many options for solving difficult problems.

An important part of the PV-WAVE:Signal Processing Toolkit is the PV-WAVE platform. Several fundamental signal processing functions already exist in PV-WAVE: including fast Fourier transform functions, numerical optimization functions, matrix manipulation functions, and functions for finding polynomial roots. The PV-WAVE:Signal Processing Toolkit greatly extends the signal processing capabilities of PV-WAVE through a combination of additional signal processing routines, and the ability to extend the functionality by customizing the source code in the Signal Processing Toolkit to meet your needs.

Purpose of this Chapter

The purpose of this chapter is to establish terminology and provide a brief overview of the functionality of the PV-WAVE:Signal Processing Toolkit. Examples in this chapter demonstrate how the Signal Processing Toolkit functions can be used together to solve signal processing problems. It is assumed that you have a basic working knowledge of signals and systems, including linear systems, transform analysis of linear systems (Fourier and *z*-transforms) and filtering.

Where appropriate, outside sources are cited, and full bibliographic entries are listed in Appendix A, *Bibliography*. In addition, the section *Background Reading* on page 29 is included in this chapter for those wishing to explore in greater detail the signal processing topics discussed in this manual.

Notation and Conventions Used in this User's Guide

The standard notation in signal processing texts uses lower-case letters for time-domain signals and upper-case for frequency-domain signals.

Use of Upper and Lower-Case Letters in This Manual

This manual follows the standard signal processing notation in the function discussions and descriptions; however, in all code examples, the PV-WAVE convention for capitalization is followed.

To illustrate the notation used in this manual, let's look at the calling sequence and discussion for the FIRFILT function. The FIRFILT calling sequence containing the filter structure $H(z)$ and the input array to be filtered, x, is as shown.

```
result = FIRFILT(h, x)
```

While the calling sequence uses h (lower-case) to represent the filter structure, the discussion uses the standard notation for signal processing, H (upper-case).

Frequency Normalization in the Signal Processing Toolkit

All Signal Processing Toolkit functions use normalized frequencies for ease in manipulation. The frequency normalization used results in a normalized Nyquist frequency equal to one.

TIP Sometimes it is preferable to show actual frequency values on an output plot. This can be easily accomplished by multiplying the normalized axis by the actual Nyquist frequency when setting the plot parameters.

Starting the Signal Processing Toolkit

If PV-WAVE isn't already installed on your system, install it first. Once PV-WAVE is installed, you're ready to start PV-WAVE so you can enter commands or initialize the Signal Processing Toolkit.

NOTE For information on installing the Signal Processing Toolkit, refer to the installation booklet inside the CD-ROM case.

Starting PV-WAVE under Windows NT

Start PV-WAVE by clicking the PV-WAVE Console icon in the PV-WAVE Program Group.

After a brief pause, the PV-WAVE Console window appears displaying the prompt:

```
WAVE>
```

At this prompt, PV-WAVE is ready for you to enter commands or initialize the Signal Processing Toolkit as described on page 5.

Starting PV-WAVE under Windows 95

Start PV-WAVE with the **Start** button. Select **Start=>Programs=> PV-WAVE 6.0=>PV-WAVE Console**

After a brief pause, the PV-WAVE Console window appears displaying the prompt:

```
WAVE>
```

At this prompt, PV-WAVE is ready for you to enter commands or initialize the Signal Processing Toolkit as described on page 5.

Starting PV-WAVE under UNIX

Start PV-WAVE by entering the following command at your UNIX system prompt:

(UNIX) `wave`

The command line prompt, `WAVE>` appears in your window.

At this prompt, PV-WAVE is ready for you to enter commands or initialize the Signal Processing Toolkit as described below.

Starting the Signal Processing Toolkit from the WAVE> Prompt

On all platforms, you can start the Signal Processing Toolkit from the `WAVE>` prompt, by doing the following:

- At the `WAVE>` prompt, enter the following commands to load and initialize the PV-WAVE:IMSL Mathematics, PV-WAVE:IMSL Statistics and the PV-WAVE:Signal Processing Toolkit:

```
WAVE> @math_startup

WAVE> @stat_startup

WAVE> @sigpro_startup
```

- Once you see the following message, you are ready to use the PV-WAVE:Signal Processing Toolkit.

```
PV-WAVE:Signal Processing Toolkit is Initialized.
```

Stopping the Signal Processing Toolkit

If the PV-WAVE:Signal Processing Toolkit is loaded and you want to exit the Signal Processing Toolkit, perform the following procedure:

- At the `WAVE>` prompt, enter the following command to unload the PV-WAVE:Signal Processing Toolkit.

```
WAVE> @sigpro_unload
```

Unloading returns your system to the state it was in before using the Signal Processing Toolkit by doing the following three things:

- It unloads the PV-WAVE:Signal Processing Toolkit functions from memory.
- It returns the Signal Processing Toolkit license to the license manager, freeing the license up for others to use.
- And it deletes all common variables in SIGPRO_COMMON.

Running the Signal Processing Toolkit Test Suite

A test suite is included with the PV-WAVE:Signal Processing Toolkit which may be used to accomplish several things. First, running the test suite is an easy way to verify that the Signal Processing Toolkit is installed properly. Second, you can look at the test suite code which provides examples of Signal Processing Toolkit functionality. And third, the test suite provides a quick, demonstration of the graphic capabilities of the Signal Processing Toolkit.

The only requirement for running the test suite is that the PV-WAVE:Signal Processing Toolkit must first be installed. After that is done, all that is necessary is to enter the following command at the `WAVE>` prompt:

```
WAVE> SIGPRO_TEST
```

The tests take approximately 10 minutes to run depending on your machine and the amount of time you spend observing each of the tests which require a keystroke to continue.

Signals and Systems Used in Signal Processing

In mathematical terms, signals are functions of an independent variable, which may be continuous or discrete. By convention, the independent variable is referred to as “time.” Continuous-time signals are termed analog signals, while discrete-time signals are called digital. Discrete-time signals, also called sequences, are typically obtained by sampling continuous-time signals.

The PV‑WAVE:Signal Processing Toolkit functions are used to process discrete-time or digital signals. Within the Signal Processing Toolkit, discrete-time signals are represented as arrays of numerical values that have been sequentially sampled at specific time intervals. In this manual, signals are referred to using the sequence notation:

$$x(n), n = 0, ..., N-1 \quad ,$$

the same notation for accessing the elements of an array in PV‑WAVE.

“Signal processing” refers to applying a function

$$T(\cdot)$$

to a signal $x(n)$ to obtain a “processed” signal, $y(n) = T(x(n))$. The function

$$T(\cdot)$$

is referred to as the system transfer function. The Signal Processing Toolkit provides a large class of routines that design and apply the transfer functions used most often in digital signal processing.

System Transfer Models and the Digital Filter Data Structure

The definition of a system transfer function is very broad. By far, the most commonly used transfer functions are those that are linear and time-invariant.

Linear System Models

Linear and time-invariant systems are represented by the convolution operation

$$y(n) = \sum_{k=-\infty}^{\infty} x(k)h(n-k) \quad ,$$

where the sequence $h(n)$ is the impulse response of the system. The class of linear time-invariant systems is very large. Among all such systems, the one defined by the linear constant-coefficient difference equation (EQ1) is central to practical applications of signal processing.

$$\begin{aligned} y(n) = {} & b_0x(n) + b_1x(n-1) + \ldots + b_Mx(n-M) \\ & - a_1y(n-1) - a_2y(n-2) - \ldots - a_Ny(n-N) \end{aligned} \qquad \text{(EQ 1)}$$

The z-transform of this difference equation is given by

$$Y(z) = H(z)X(z),$$

where $X(z)$ and $Y(z)$ are the z-transforms of the sequences $x(n)$ and $y(n)$, respectively, and $H(z)$ is the rational transfer function

$$H(z) = \frac{B(z)}{A(z)} = \frac{b_0 + b_1z^{-1} + \ldots + b_Mz^{-M}}{1 + a_1z^{-1} + \ldots + a_Nz^{-N}} \qquad \text{(EQ 2)}$$

The function $H(z)$ is simply referred to as a digital filter. If the filter coefficients satisfy $a_n = 0$ for $n = 1, 2, ..., N$, then the filter has finite impulse response (FIR). Otherwise, if $a_n \neq 0$ for $n > 1$, the filter has infinite impulse response (IIR).

There are many ways to rearrange the difference equation (EQ 1) and transfer function (EQ 2). Standard canonic forms, in addition to the transfer function form (EQ 2), include the zero-pole-gain (first order cascade) form, second order cascade form, partial fraction (first order parallel) form, second order parallel form, and state-space form (for single input and single output). All of these canonic forms are theoretically equivalent to the transfer function form (EQ 2).

In practice, with finite precision arithmetic, the various canonic forms provide varying degrees of numerical precision. The PV-WAVE:Signal Processing Toolkit uses the transfer function canonic form, and all computations that use this canonic form are carried out using double-precision arithmetic. If you require any of the other canonic forms for their application, it is possible to transfer back and forth between the various forms using the fundamental polynomial manipulation functions (see *Polynomial Manipulation* on page 25) provided in the Signal Processing Toolkit

Filter Data Structures

Because the rational transfer function model

$$H(z) = \frac{B(z)}{A(z)} = \frac{b_0 + b_1 z^{-1} + \ldots + b_M z^{-M}}{1 + a_1 z^{-1} + \ldots + a_N z^{-N}}$$

is so important to classical signal processing, the PV-WAVE:Signal Processing Toolkit uses a data structure to represent the rational transfer functions and simplify your task of keeping track of the numerator and denominator coefficients.

Routines for accessing the filter structure information are summarized in the following table. For detailed information, see Chapter 2, Reference.

Filter Data Structure Routines

Routine	**Description**
FILTSTR (*)	Place information into filter structure.
PARSEFILT (*)	Extract information from filter structure.

*   function uses the digital filter data structure

The basic information contained within the filter structure includes an array of real numerator coefficients, an array of real denominator coefficients, and a name string. Within the filter structure, the coefficients are represented in double precision.

All filter approximation and realization functions in the PV-WAVE:Signal Processing Toolkit use filter structures. The following example illustrates the use of a filter structure.

```
numerator = [1, 3, 3, 1]
    ; Numerator coefficients of the rational transfer function.
denominator = [1, 0, 1/3]
    ; Denominator coefficients of the rational transfer function.
h = FILTSTR(numerator, denominator, Name = 'example')
        ; Places the filter coefficients into the filter structure.
PARSEFILT, h, filtname, b, a
    ; Extracts the filter coefficients from the filter structure.
PRINT, filtname
    example
```

```
PM, b
   1.0000000
   3.0000000
   3.0000000
   1.0000000
PM, a
   1.0000000
   0.0000000
   0.3333333
```

Digital Filter Analysis Techniques

Two of the most basic techniques used in analyzing digital filters are determining the frequency response and determining the impulse response of a rational transfer function. The Signal Processing Toolkit functions that perform these operations are summarized in the following table. For detailed information, see Chapter 2, Reference.

Filter Analysis Functions

Function	**Description**
ABS (*)	Magnitude of a complex array
ARG	Phase of a complex array
FREQRESP_S (**)	Complex frequency response of analog transfer function (Bode plot)
FREQRESP_Z (**)	Complex frequency response of digital transfer function
IMPRESP	Impulse response of a filter

*   function reference found in PV-WAVE Reference
** function uses the digital filter data structure

The following example illustrates the use of the filter analysis functions by computing the frequency response of a first order Butterworth lowpass filter with a normalized frequency cutoff of 0.2049. The resulting magnitude and phase plots for the frequency response of the filter in this example code are shown in *Figure 1-1*.

```
h = FILTSTR([.25, .25],[1, -.5])
    ; Places the Butterworth filter coefficients into the filter structure.
hf = FREQRESP_Z(h, Outfreq = f)
    ; Computes the complex frequency response of the filter on the
    ; unit circle.
!P.Multi = [0, 1, 2]
    ; Produce one column of two plots.
PLOT, f, ABS(hf), Title = 'Magnitude Response'
    ; Plot the magnitude of the complex frequency response
    ; (Figure 1-1 (a)).
PLOT, f, ARG(hf), Title = 'Phase Response'
    ; Plot the phase of the complex frequency response (Figure 1-1 (b)).
```

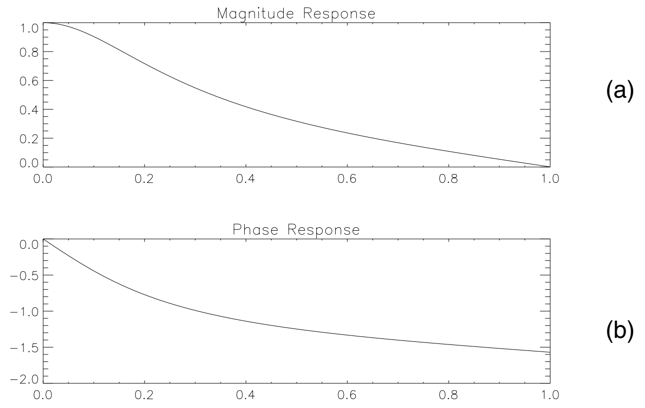


Figure 1-1 The magnitude response (a) and phase response (b) of a first order Butterworth lowpass filter.

Filter Approximation

Digital filter design problems consist of two parts, approximation and realization. This section discusses the PV‑WAVE:Signal Processing Toolkit functions for approximating digital filters. The Filter Realization section discusses the routines for realizing digital filters.

Digital filter approximation problems consist of selecting the coefficients of the rational transfer function *H*(*z*),

$$H(z) = \frac{B(z)}{A(z)} = \frac{b_0 + b_1 z^{-1} + \ldots + b_M z^{-M}}{1 + a_1 z^{-1} + \ldots + a_N z^{-N}}$$

in order to achieve some desired result when the filter is applied to a signal. All of the filter approximation routines in the PV‑WAVE:Signal Processing Toolkit return the filter coefficients in a digital filter data structure.

Classical FIR and IIR Filter Approximation

Classical FIR and IIR filter approximation problems concern the approximation of the ideal lowpass, highpass, bandpass, and bandstop filters as illustrated in *Figure 1-2*.

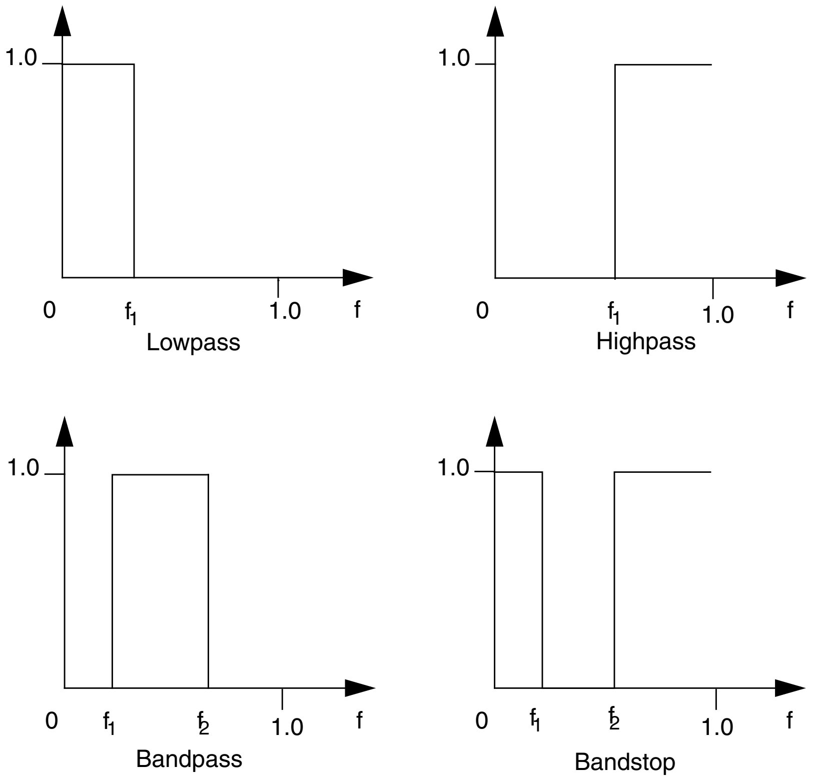


Figure 1-2 Ideal filters used in classical FIR and IIR filter approximation.

A summary of PV-WAVE:Signal Processing Toolkit functions for solving classical filter design problems is listed in the following table. FIR and IIR filter approximations use separate functions in the Signal Processing Toolkit. For detailed information, see Chapter 2, Reference.

Classical Filter Approximation Functions

Function	**Description**
BILINTRANS (*)	Bilinear transform
FIRDESIGN (*)	FIR lowpass, highpass, bandpass, bandstop filter design
FIRLS (*)	FIR multiple bandpass FIR filter design
FIRWIN	FIR window functions
FREQTRANS (*)	IIR filter frequency transformation
FREQTRANSDESIGN	IIR filter frequency transformation design for multiple bandpass IIR filter design
IIRDESIGN (*)	IIR Butterworth, Chebyshev I, Chebyshev II, and elliptic filter design
IIRORDER	IIR filter order estimation

* function uses the digital filter data structure

FIR Filter Approximation

The classical approach to FIR filter design uses window functions. This approach first determines the inverse Fourier transform of the ideal filter frequency response

$$H_{ideal}(e^{j\pi f}) = \begin{cases} 1, & f \text{ in passband} \\ 0, & f \text{ in stopband} \end{cases}$$

and then multiplies this response by an appropriate window function.

The following example illustrates the design of a windowed bandpass filter. *Figure 1-3* shows the resulting filter frequency response.

```
w = FIRWIN(55, /Blackman)
      ; Computes a window sequence.
h = FIRDESIGN(w, 0.33, 0.66, /Bandpass)
      ; Design a windowed FIR filter with normalized cutoff frequencies
      ; of 0.33 and 0.66.
hf = FREQRESP_Z(h, Outfreq = f)
PLOT, f, ABS(hf)
      ; Plot the magnitude of the filter frequency response (see Figure 1-3).
```

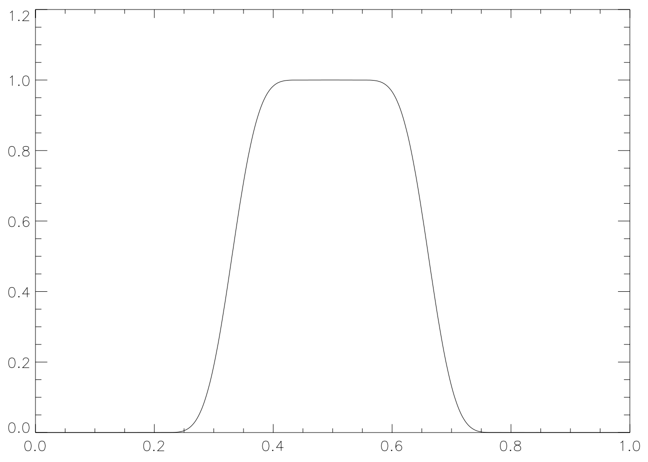


Figure 1-3 Windowed bandpass filter frequency response.

IIR Filter Approximation

One of the key features of the PV-WAVE:Signal Processing Toolkit is the flexibility of IIR filter design which results from the approach used in applying transformations in IIR filter design. This approach enables you to design all of the filters in the classical approach, and has the added advantage of simplifying the design of multiple bandpass filters.

The classical approach to IIR filter design starts with an analog lowpass filter prototype and then applies various frequency transformations to arrive at the desired digital filter. There are two approaches to applying transformations to an analog lowpass filter prototype as illustrated in *Figure 1-4*. The top path in the figure illustrates the standard approach taken in IIR filter design, and the bottom path illustrates the alternate approach used in the PV-WAVE:Signal Processing Toolkit.

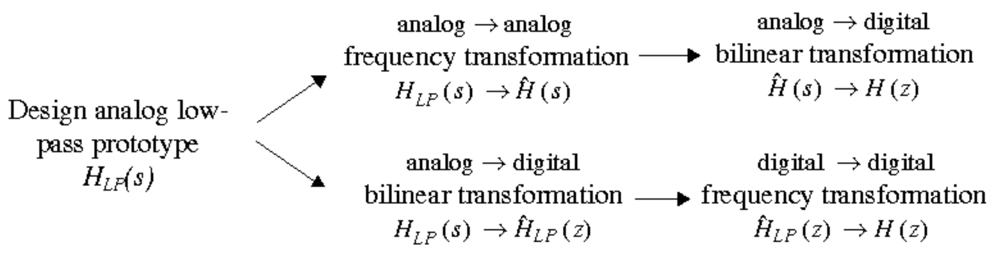

Design analog low-pass prototype $H_{LP}(s)$
analog → analog frequency transformation $H_{LP}(s) \to \hat{H}(s)$
analog → digital bilinear transformation $\hat{H}(s) \to H(z)$
analog → digital bilinear transformation $H_{LP}(s) \to \hat{H}_{LP}(z)$
digital → digital frequency transformation $\hat{H}_{LP}(z) \to H(z)$

Figure 1-4 Transformation application approaches. The top path represents the standard approach used by signal processors. The bottom path represents the approach used by the Signal Processing Toolkit.

The Standard IIR Approximation Approach

The approach illustrated in the top path of *Figure 1-4* is the approach most often employed by signal processors. This standard approach first transforms the analog lowpass filter prototype $H_{LP}(s)$ into an appropriate lowpass, highpass, bandpass, or bandstop filter

$$\hat{H}(s),$$

and then uses the bilinear transform to obtain a digital filter $H(z)$.

The PV-WAVE:Signal Processing Toolkit Approach

The approach used in the Signal Processing Toolkit is shown in the bottom path in *Figure 1-4*. This alternate approach can design all of the filters available using the standard approach, but has the added advantage of simplifying multiple bandpass filter design. This approach first transforms the analog lowpass filter prototype $H_{LP}(s)$ into a digital lowpass filter prototype

$$\hat{H}_{LP}(z)$$

using the bilinear transform. Then a frequency transformation is applied to obtain an appropriate lowpass, highpass, bandpass, or bandstop digital filter $H(z)$.

The following example illustrates the ease in designing a multiple bandpass elliptic filter using the PV-WAVE:Signal Processing Toolkit.

```
!P.Multi = [0, 1, 2]
hlp = IIRDESIGN(3, 0.5, 0.1, 0.1, /Ellip)
    ; Design a digital lowpass filter prototype with a frequency band
    ; edge of 0.5.
```

```
hlpf = FREQRESP_Z(hlp, Outfreq = f)

PLOT, f, ABS(hlpf), $
    Title = 'Lowpass Prototype Filter'
        ; Plot the magnitude frequency response of the lowpass filter
        ; prototype.

p = FREQTRANSDESIGN([.1, .3, .4, .6, .8])
    ; Design a frequency transformation for a multiple bandpass filter
    ; with frequency band edges of 0.1, 0.3, 0.4, 0.6, and 0.8.

hbp = FREQTRANS(hlp, p)
    ; Apply the frequency transformation to the lowpass filter prototype.

hbpf = FREQRESP_Z(hbp, Outfreq = f)

PLOT, f, ABS(hbpf), $
    Title = 'Multiple Bandpass Filter'
        ; Plot the magnitude response of the multiple bandpass filter.
```

In *Figure 1-5*, (a) shows the magnitude frequency response of the original lowpass filter, and (b) shows the response after applying the frequency transformation to obtain the multiple bandpass filter.

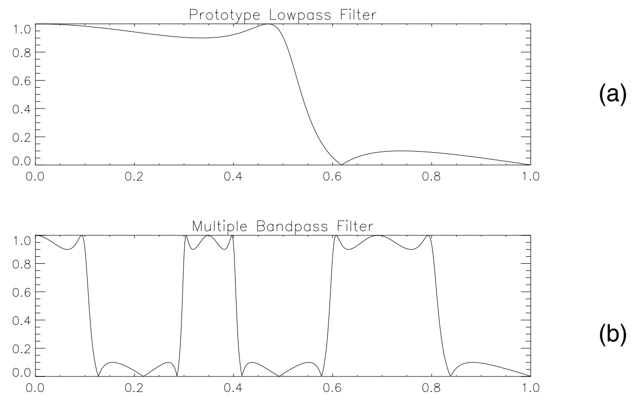


Figure 1-5 A multiple bandpass filter (b) designed from a prototype digital lowpass filter (a) using the Signal Processing Toolkit approach to IIR filter design.

Advanced and Multirate Filter Approximation

The PV-WAVE:Signal Processing Toolkit also includes several advanced filter approximation functions and multirate filter approximation functions.

The advanced filter approximation functions produce filters that are optimal subject to various constraints, such as least-squares error criteria, Chebyshev error criteria, and moment preserving criteria.The least-squares filter approximation functions shown in the following table can also be used to obtain filters that interpolate a given set of impulse or frequency response values. For detailed information, see Chapter 2, Reference.

Advanced Filter Approximation Functions

Function	**Description**
FIRLS (*)	Frequency domain least-squares FIR filter design
IIRLS (*)	Frequency and time domain least-squares filter design
REMEZ (*)	Optimal Chebyshev error FIR filter design (Parks-McClellan algorithm)
SGFDESIGN (*)	Optimal Savitsky-Golay FIR filter design

* function uses digital filter data structure

The next table shows the list of multirate filter approximation functions that produce filters which are used in standard multirate filtering operations such as decimation and interpolation. For detailed information, see Chapter 2, Reference.

Multirate Filter Approximation Functions

Function	**Description**
FILTDOWNDESIGN (*)	Decimation filter design
FILTUPDESIGN (*)	Interpolation filter design
QMFDESIGN (*)	Quadrature mirror filter design

* function uses digital filter data structure

Filter Realization

The digital filter realization problem is concerned with numerically computing the output signal $y(n)$ via the difference equation

$$y(n) = b_0x(n) + b_1x(n-1) + \ldots + b_Mx(n-M)$$
$$- a_1y(n-1) - a_2y(n-2) - \ldots - a_Ny(n-N) \ ,$$

with input signal $x(n)$. The PV-WAVE:Signal Processing Toolkit includes functions for realizing FIR, IIR and multirate digital filters.

Standard FIR and IIR Filter Realization

Functions for realizing FIR and IIR filters are summarized in the following table. For detailed information, see Chapter 2, Reference.

FIR and IIR Filter Realization

Function	**Description**
FILTER(*)	Basic FIR and IIR filter realization
FIRFILT (*)	Convolution and FFT-based filter realization of FIR filters
IIRFILT (*)	Causal and anti-causal (forward-backward) IIR filter realization

* function uses digital filter data structure

FILTER implements both FIR and IIR filters in a fashion that is transparent to you, and it is the workhorse used in most applications. The FIRFILT and IIRFILT functions provide you with increased control over how the FIR and IIR filters are realized.

The following example shows a typical application of the FILTER function.

```
!P.Multi = [0, 1, 1]

t = FINDGEN(1024)

s1 = SIN(0.6*t)

s2 = SIN(1.2*t)
   ; Generates two narrow band signals.

x = s1 + s2
   ; Combines the two signals.

f = FINDGEN(512)/511
   ; Generate the abscissa values for the normalized frequency.

PLOT, f, (ABS(FFTCOMP(x, /Complex)))(0:512), $
   Title = 'Original'
      ; Plot the magnitude frequency response of the combined signal.
```

```
h = FIRDESIGN(101, 0.1, 0.25, /Bandpass)
    ; Approximate a bandpass filter to isolate the first signal.
y = FILTER(h, x)
    ; Apply the filter to the signal.
PLOT, f, (ABS(FFTCOMP(y, /Complex)))(0:512), $
    Title = 'Filtered'
        ; Plot the magnitude frequency response of the filtered signal.
```

The original combined narrow band signal is shown in (a) and the results of the example code using the FILTER function are shown in (b) of *Figure 1-6*.

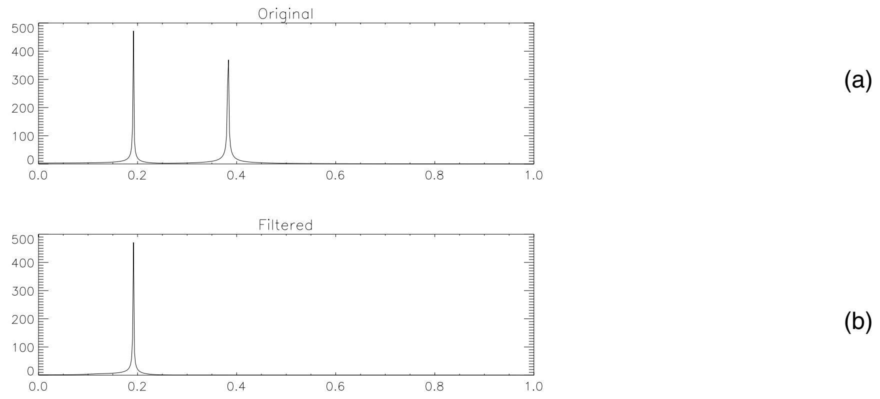


Figure 1-6 The result (b) of filtering a combined narrow band signal (a) using the FILTER function.

The next example illustrates how IIRFILT may be used to perform causal and anti-causal filtering.

```
!P.Multi = [0, 1, 2]
x = ((INDGEN(1024)+64) MOD 256) GT 128
    ; Generate a square wave.
h = IIRDESIGN(5, 0.25, 0.01, 0.01, /Ellip)
    ; Design an elliptical lowpass filter.
PLOT, IIRFILT(h, x), Title = 'Causal Filtering'
        ; Plot the causal filtering of the square wave.
PLOT, IIRFILT(h, x, /Forward_back), Title = 'Anti-Causal Filtering'
        ; Plot the anti-causal (forward-backward) filtering of the
        ; square wave. Notice that the result is symmetric, and there
        ; is no phase distortion.
```

Figure 1-7 (a) shows the causal (forward) filter results and (b) shows the anti-causal (forward-backward) filter results obtained by applying the IIRFILT function to the square wave.

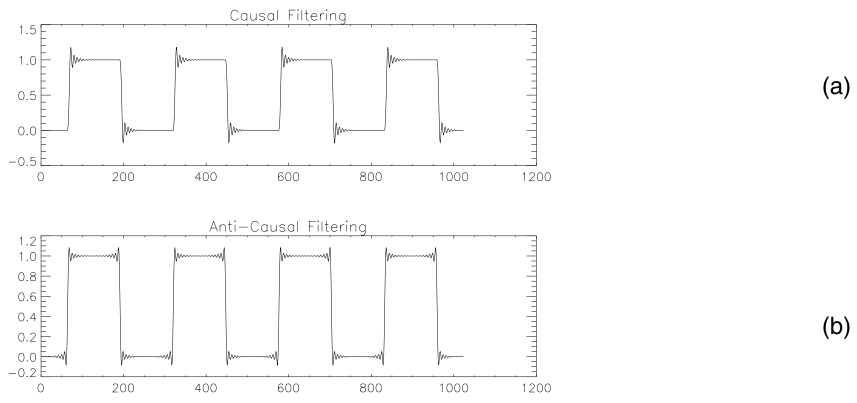


Figure 1-7 Example of causal (a) and anti-causal (b) filtering of a square wave using IIRFILT.

Multirate Filter Realization

Functions and procedures for realizing multirate filters are summarized in the following table. For detailed information, see Chapter 2, Reference.

Multirate Filter Realization Routines

Routine	Description
FILTDOWN (*)	Decimation filter
FILTUP(*)	Interpolation filter
QMF (*)	Quadrature mirror filter

* routine uses digital filter data structure

FILTDOWN and FILTUP can be combined to achieve most multirate signal processing techniques. The commonly used quadrature mirror filter operation is provided in the QMF procedure.

Transforms and Spectrum Analysis

One of the most important operations in signal processing is transforming a signal from one domain to another to analyze and extract information. The PV‑WAVE:Signal Processing Toolkit provides several functions and procedures for the Fourier and wavelet analysis of a signal. These routines are listed in the following table. For detailed information, see Chapter 2, Reference.

Transforms and Spectrum Analysis Routines

Routine	Description
DCMPLXFFT	Double-precision complex fast Fourier transform
FFTCOMP	Fast Fourier transform
FFTINIT	Fast Fourier transform initialization
SPECTROGRAM	Spectrogram or short-time Fourier transform analysis
SPECTRUM	Power spectrum analysis (power spectral density)
WAVELET (*)	Wavelet transform

* function uses digital filter data structure

The fundamental computational tool in signal processing is the fast Fourier transform (FFT). The PV‑WAVE:Signal Processing Toolkit provides several methods of computing the FFT.

The SPECTRUM and SPECTROGRAM functions provide essential tools for Fourier analysis of stationary and non-stationary signals. Non-stationary signals can also be effectively analyzed using the WAVELET function, which computes the wavelet transform of a signal using compactly supported orthonormal wavelets.

In the following example, SPECTROGRAM is used on a file containing a signal of a human voice.

(UNIX) `To open the file on a UNIX system:`

```
OPENR, u, GETENV('VNI_DIR')+ $
   '/sigpro-1_1/test/voice.dat', /Get_Lun
```

(OpenVMS) `To open the file on an OpenVMS system:`

```
OPENR, u, GETENV('VNI_DIR')+ $
   '[SIGPRO-1_1.TEST]VOICE.DAT', /Get_Lun
```

(Windows) `To open the file on a Windows system:`

```
OPENR, u, GETENV('VNI_DIR')+ $
   '\sigpro-1_1\test\voice.dat', /Get_Lun
```

```
x = BYTARR(7519)

READU, u, x

CLOSE, u

xs = 400

ys = 400

WINDOW, XSize = xs, YSize = ys

PLOT, x, Position = [0, .5, 1, 1], $
   XStyle = 5, YStyle = 5, /Normal
      ; Plot the original signal.

mat = SPECTROGRAM(x, 256)
   ; Compute the spectrogram of the original signal.

TVSCL, HIST_EQUAL(CONGRID( $
   ALOG10(mat > 1.e-5), xs, ys/2))
      ; Display the spectrogram as an image.
```

The original voice signal, a non-stationary signal with amplitude and frequency changing over time is shown in (a) of *Figure 1-8*. The resulting spectrogram image shown in (b) of *Figure 1-8* is the visual representation of the power spectral density of the signal versus time. The magnitude of the power spectrum image in (b) is indicated by the grey (or color) scale.

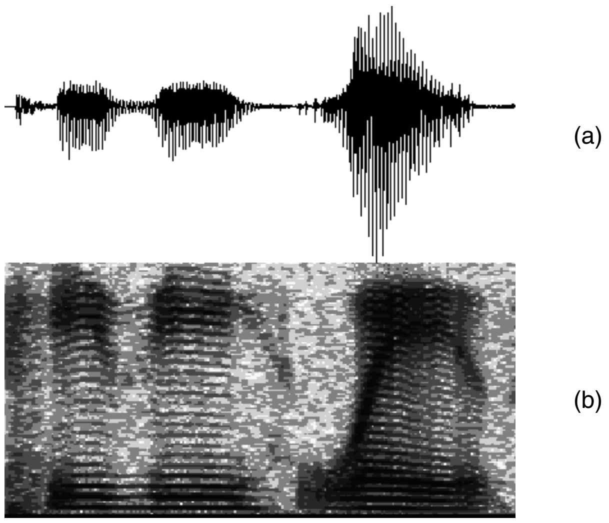


Figure 1-8 A voice signal (a) is processed with SPECTROGRAM and displayed in PV‑WAVE as an image.

Statistical Signal Processing

Just as linear shift-invariant systems and rational transfer models are central to classical signal processing, stationary signals and Toeplitz covariance matrix models are central to statistical signal processing. An extensive selection of routines is provided for solving fundamental statistical signal processing problems.

The PV-WAVE:Signal Processing Toolkit provides a complete suite of routines for manipulating Toeplitz matrix equations: the JURYRC, LEVCORR and LEVDURB procedures, and the TOEPSOL function. These routines and the other various statistical signal processing functions are summarized in the following table. For more detailed information, see Chapter 2, Reference.

Statistical Signal Processing Routines

Routine	**Description**
JURYRC	Jury (reflection coefficient) algorithm
LEVCORR	Auto-correlation sequence computation from factored Toeplitz forms
LEVDURB	Levinson-Durbin algorithm for factoring Toeplitz matrices
TOEPSOL	Levinson's algorithm for solving Toeplitz linear equations
FIRLS (*)	FIR Wiener filter design
IIRLS (*)	Prony's and frequency-sampling methods
LPC (*)	Linear prediction coefficients
RANDOM	Random number generation
RANDOMOPT	Random number generation control

* function uses digital filter data structure

The optimal linear phase FIR Wiener filter design problem is solved using the FIRLS function. A connection between the transfer function models of classical signal processing and the stationary random signal models of statistical signal processing is provided by Prony's method which is part of the IIRLS function.

The most important signals in theoretical statistical signal processing are normal random variables and quadratic forms of normal random variables. The PV-WAVE RANDOM and RANDOMOPT functions provide the basic tools for generating such random variables. The PV-WAVE reference pages for these routines are reproduced in this manual for your convenience.

Polynomial Manipulation

The most fundamental function in classical signal processing is the rational transfer function

$$H(z) = \frac{B(z)}{A(z)} = \frac{b_0 + b_1 z^{-1} + \ldots + b_M z^{-M}}{1 + a_1 z^{-1} + \ldots + a_N z^{-N}} \;,$$

which is a ratio of two polynomials in z^{-1}. The ability to manipulate polynomials is therefore one of the most fundamental operations in signal processing. The PV‑WAVE:Signal Processing Toolkit provides a rich set of polynomial manipulation functions that greatly enhance your ability to extend the functionality of the Signal Processing Toolkit.

The polynomial manipulation functions belong to one of two classes. Those that manipulate standard polynomials of the form

$$c(x) = c_0 + c_1 x + c_2 x^2 + \ldots + c_N x^N$$

are summarized in the following table. For detailed information, see Chapter 2, Reference.

Standard Polynomial Manipulation Routines

Routine	**Description**
CONVOL1D	One dimensional convolution
DBLPOLY	Polynomial function evaluation
PAIRCONJ	Pair conjugate complex numbers
PAIRINV	Pair reciprocal complex numbers
P_DEG	Numerical determination of polynomial degree
P_DIV	Polynomial division
P_MULT	Polynomial multiplication
P_SQRT	Polynomial spectral factorization
P_SUM	Polynomial sum
ROOT2POLY	Compute coefficients of polynomial with specified roots
ZEROPOLY	Polynomial root finding

Those functions that manipulate polynomials in z^{-1} of the form

$$D(z) = d_0 + d_1 z^{-1} + d_2 z^{-2} + \ldots + d_N z^{-N}$$

are summarized in the following table. For more information, see Chapter 2, Reference.

Functions for Polynomials in z^{-1}

Function	**Description**
P_STAB	Polynomial stabilization
SCHURCOHN	Schur-Cohn stability test

A common signal processing problem is having to simultaneously manipulate polynomials in both z and z^{-1}. For example, the magnitude frequency response of a real-coefficient FIR filter

$$B(z) = b_0 + b_1 z^{-1} + b_2 z^{-2} + \ldots + b_M z^{-M}$$

may be found by evaluating $B(z)B(z^{-1})$ for values of z on the unit circle.

The REVERSE function in PV-WAVE is useful for manipulating polynomials in z and z^{-1}. The mathematical equivalent of reversing the coefficients of a polynomial is given by

$$\tilde{B}(z) = z^{-M} B(z^{-1}) = b_0 z^{-M} + b_1 z^{-M+1} + \ldots + b_M \quad .$$

One way to compute $D(z) = B(z)B(z^{-1})$ is to compute

$$B(z)\tilde{B}(z) = z^{-M} B(z) B(z^{-1}) ,$$

which has the same coefficient ordering as $D(z)$.

Using PV-WAVE:Signal Processing Toolkit commands, the operation

$$D(z) = B(z)B(z^{-1})$$

is simply

```
d = P_MULT(b, REVERSE(b))
```

where the arrays `b` and `d` contain the coefficients of the polynomials $B(z)$ and $D(z)$, respectively.

Optimization

Most signal processing design problems involve finding the solution to optimization problems. For example, the routines FILTUPDESIGN, FIRLS, IIRDESIGN, IIRLS, REMEZ and SGFDESIGN are all optimal filter design solutions under various constraints and error functionals. In many practical applications, however, custom design of optimal signal processing operations is desirable.

PV-WAVE provides many numerical optimization functions for solving advanced signal processing problems. Because optimization problems occur so often in signal processing, the reference pages of the optimization routines available in PV-WAVE are included in this manual for your convenience. These optimization functions are listed in the following table. For detailed information, see Chapter 2, Reference.

PV-WAVE Optimization Functions

Function	Description
FMIN	Univariate unconstrained minimization
FMINV	Multivariate unconstrained minimization
INTFCN	Numerical integration or quadrature
LINPROG	Linear programming constrained minimization
NLINLSQ	Nonlinear least-squares unconstrained minimization
NONLINPROG	Nonlinear programming nonlinearly constrained minimization
NORM	Computes various array norms
QUADPROG	Quadratic programming constrained minimization

PV-WAVE functions for performing numerical integration and computing array norms, such as INTFCN and NORM are also included in this manual. These functions are often used in conjunction with optimization routines to evaluate error functionals.

Plotting and Signal Generation Routines

PV-WAVE provides a rich set of plotting routines. Those plotting routines are augmented in the PV-WAVE:Signal Processing Toolkit with some specialized plotting routines. The Signal Processing Toolkit function SIGNAL, for instance allows you to easily generate any of eight commonly used signals. The additional plotting and signal generation routines provided in the Signal Processing Toolkit are listed in the following table. For more information, see Chapter 2, Reference.

Plotting and Signal Generation Routines

Routine	Description
OPLOTCOMB	Comb plot over a previously drawn plot
PLOTCOMB	Comb plot
PLOTZP	Zero-pole plot
REFLINES	Reference lines to indicate specific axis values
SIGNAL	Generates commonly used signals

Functional Overlaps

During the development of the PV-WAVE:Signal Processing Toolkit some functional overlap with PV-WAVE was introduced. The following areas have been identified as having overlap.

- Fourier Transforms

  PV-WAVE already has the function FFTCOMP for computing the real or complex FFT, but since a double-precision complex FFT was needed for PV-WAVE:Signal Processing Toolkit, the function DCMPLXFFT has been implemented.
- Polynomial Evaluation

  The PV-WAVE function POLY performs evaluation of polynomials, however since a double-precision complex evaluation was needed for PV-WAVE:Signal Processing Toolkit, the function DBLPOLY was devised.
- DIGITAL_FILTER

  DIGITAL_FILTER is an older PV-WAVE function which, like the Signal Processing Toolkit FILTER and FIRWIN functions uses a Kaiser window to filter an input signal. Although DIGITAL_FILTER is available, as a signal processor you should use the functions FIRWIN and FILTER to perform data filtering with a Kaiser window.

Background Reading

If you wish to explore the various signal processing topics discussed in this manual in greater detail, the following list of suggested references is a good starting point. Full bibliographic entries for this suggested reading list are found in Appendix A, *Bibliography*.

Signals and Systems

Signals and systems theory (including models and analysis) is the basic prerequisite for digital signal processing. Standard texts for these topics include Hamming, 1989; Jackson, 1991; Oppenheim, Willsky, and Young, 1983; Gabel and Roberts, 1987.

Filter Approximation and Realization

Many standard texts are available that discuss basic and advanced digital filtering topics, including Oppenheim and Schafer, 1989; Parks and Burrus, 1987; Proakis and Manolakis, 1992; and Roberts and Mullis, 1987.

Good references on multirate filtering and signal processing include Vaidyanathan, 1993; and Akansu and Haddad, 1992. For a standard application requiring multiple bandpass filters, see Jayant and Noll, 1984, p. 641.

Transforms and Spectrum Analysis

Most introductory digital signal processing texts include chapters on the fast Fourier transform, including those by Oppenheim and Schafer, 1989; Parks and Burrus, 1987; Proakis and Manolakis, 1992; and Roberts and Mullis, 1987.

Standard references on spectrum analysis and spectrograms include Kay, 1987; Kay, 1993; Marple, 1987; and Nawab and Quatieri, 1988; in addition to such standard signal processing texts by Oppenheim and Schafer, 1989; Parks and Burrus, 1987; Proakis and Manolakis, 1992; and Roberts and Mullis, 1987.

References that discuss wavelets as they relate to signal processing problems include Rioul and Vetterli, 1991; and Vaidyanathan, 1993.

A discussion on wavelets as they relate to signal processing problems can be found in Vaidyanathan, 1993.

More mathematical treatments of wavelet theory can be found in Daubechies, 1992; and Chui, 1992.

Statistical Signal Processing

Recent texts on statistical signal processing include those by Scharf, 1991; Kay, 1993; Porat, 1994; and Therrien, 1992.

Good coverage of the various Toeplitz matrix operations that are commonly used in statistical signal processing may be found in Roberts and Mullis, 1987, Chapter 11.

Polynomial Manipulation

Thorough coverage of polynomial manipulation and its importance to efficient signal processing algorithms may be found in Blahut, 1985.

Optimization

A standard text on the basics of solving optimization problems is by Luenberger, 1984.

2

CHAPTER

Reference

This chapter describes each of the procedures and functions of the PV-WAVE:Signal Processing Toolkit. These descriptions are arranged in alphabetical order by routine name.

For a list of PV-WAVE routines that can be used in signal processing applications, see Appendix B, *Related Routines*.

ARG Function

Computes the phase angle, in radians, of a complex scalar or array.

Usage

result = ARG(*z*)

Input Parameters

z — A complex scalar or array.

Returned Value

result — A double-precision scalar or array. The range of the returned values is between $-\pi$ and π.

Keywords

None.

Discussion

The function divides the imaginary part of *z* by the real part of *z* and computes the arctangent of the quotient.

For a given a complex number of the form

$$z = x + jy = re^{j\phi} ,$$

the function computes ϕ as

$$\phi = \text{atan}(y/x).$$

Example

ARG is used to compute the polar form of the complex number $z = 3 + j4$.

```
z = COMPLEX(3, 4)
    ; Create a complex variable.
r = ABS(z)
phi = ARG(z)
PRINT, r, phi
   5.00     0.927295
```

See Also

In the *PV-WAVE Reference*:

ABS, ATAN, COMPLEX

BILINTRANS Function

Computes the bilinear transform of an analog transfer function.

Usage

result = BILINTRANS(*h* [, *k*])

Input Parameters

h — A valid analog filter structure defined as the ratio of two polynomials in positive powers of *s*.

k — (optional) A multiplier constant. (Default: *k* = 1)

Returned Value

result — A digital filter structure containing the transfer function made of a ratio of polynomials in negative powers of *z*.

Keywords

Newname — A scalar string specifying a name for the new filter structure. If not used, the new filter structure has the same name as the old one.

Discussion

For a given analog transfer function of the form

$$H_a(s) = \frac{B_a(s)}{A_a(s)} = \frac{b_{0_s} + b_{1_s}s + b_{2_s}s^2 + \ldots + b_{M_s}s^M}{a_{0_s} + a_{1_s}s + a_{2_s}s^2 + \ldots + a_{N_s}s^N}$$

where B_a and A_a are polynomials in positive powers of *s*, BILINTRANS performs a bilinear transformation

$$H_d(z) = H(s)\big|_{s = k[(z-1)/(z+1)]} = \frac{B_d(z)}{A_d(z)}$$

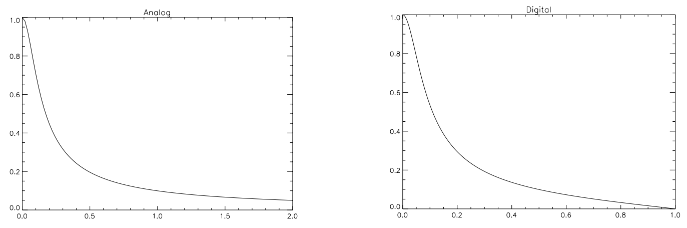
to obtain a digital rational transfer function $H_d(z)$ in negative powers of *z*.

$$_d(z) = \frac{b_{0_z} + b_{1_z}z^{-1} + b_{2_z}z^{-2} + \ldots + b_{M_z}z^{-M}}{a_{0_z} + a_{1_z}z^{-1} + a_{2_z}z^{-2} + \ldots + a_{N_z}z^{-N}}$$

Example

In this example we call BILINTRANS with a simple lowpass filter and plot the frequency response of both the analog and digital form of the filter.

```
b = [0.1]

a = [-0.1, 1.0]

h= FILTSTR(b, a)
   ; Define a simple analog lowpass filter H(s) = 0.1/(s – 0.1).

omega = FINDGEN(100)/50.0

PLOT, omega, ABS(FREQRESP_S(h, $
   COMPLEX(FLTARR(100), omega))), Title = 'Analog'
      ; Plot the frequency response of the analog transfer function.
      ; See Figure 2-1 (a).

hd = BILINTRANS(h)
   ; Transform the analog transfer function to digital.

hdresp = FREQRESP_Z(hd, Outfreq = f)

PLOT, f, ABS(hdresp), Title = 'Digital'
   ; Plot the result. See Figure 2-1 (b).
```



(a) (b)

Figure 2-1 Frequency response of the original analog filter (a) and the digital filter (b) obtained using BILINTRANS.

See Also

FILTSTR

For Additional Information

Oppenheim and Schafer, 1989.
Parks and Burrus, 1987.
Proakis and Manolakis, 1988.
Roberts and Mullis, 1987.

CONVOL1D Function

Computes the discrete convolution of two sequences.

Usage

result = CONVOL1D(*x*, *y*)

Input Parameters

x — A one-dimensional array.

y — A one-dimensional array.

Returned Value

result — A one-dimensional array containing the discrete convolution of *x* and *y*.

Keywords

Direct — If set, the computation is performed using the direct method rather than the FFT method, regardless of the size of the arrays.

Periodic — If present and nonzero, a circular convolution is computed.

Discussion

The function CONVOL1D computes the discrete convolution of two sequences $x(k)$ and $y(k)$ given by

$$z(k) = \sum_n x(n)y(n-k) .$$

The values of $x(n)$ and $y(n-k)$ are assumed to be zero when the index is not between $[0, Nx-1]$ and $[0, Ny-1]$, respectively.

If the lengths of the sequences are small, the direct sum formula is used. Otherwise, for longer sequences the FFT is used to compute the convolution.

Example

In this example, the convolution of *x* = [1, 2, 3, 2, 1] and *y* = [2, 2, 2] is computed.

```
x = [1, 2, 3, 2, 1]
y = [2, 2, 2]
    ; Define x and y.
PM, CONVOL1D(x, y)
    2.0000000
    6.0000000
    12.000000
    14.000000
    12.000000
    6.0000000
    2.0000000
```

DBLPOLY Function

Evaluates a polynomial function in double precision using Horner's method.

Usage

result = DBLPOLY(*x*, *coefficients*)

Input Parameters

x — A scalar or array variable used to evaluate the polynomial.

coefficients — An array containing the coefficients of the polynomial which has one more element than the degree of the polynomial function.

Returned Value

result — An array with the same dimensions as the input parameter *x*, containing the polynomial function evaluated at *x*.

Keywords

None.

Discussion

DBLPOLY evaluates the polynomial function

$$c(x) = c_0 + c_1x + c_2x^2 + \dots + c_{n-1}x^{n-1} ,$$

where n is the dimension of $c(x)$, and $c_0, c_1, \dots, c_{n-1}$ are the elements of the input parameter *coefficients*.

DBLPOLY returns an array with the same dimensions as *x*.

DBLPOLY uses Horner's method to evaluate a polynomial. The routine is the same as the PV-WAVE POLY function, except when either parameter is complex. If either parameter is complex, the result is computed with double-precision, as opposed to the single precision arithmetic performed in POLY.

Example

Evaluate the polynomial $c(z) = 1 + z^3$ at $z = (1 + j)$.

```
PRINT, DBLPOLY(COMPLEX(1, 1), [1, 0, 0, 1])
    -1.00000,        2.00000
```

See Also

In the *PV-WAVE Reference*: POLY

DCMPLXFFT Procedure

Computes a complex fast Fourier transform (FFT) using double precision.

Usage

DCMPLXFFT, *r_in*, *i_in*, *r_out*, *i_out*

Input Parameters

r_in — The real part of the complex array to be transformed.

i_in — The imaginary part of the complex array to be transformed.

Output Parameters

r_out — The real part of the transformed array.

i_out — The imaginary part of the transformed array.

Keywords

Backward — If present and nonzero, an inverse FFT is computed.

Discussion

DCMPLXFFT uses the real FFT provided by FFTCOMP to compute a complex FFT in double precision.

See Also

FFTCOMP

In the *PV*‑*WAVE Reference*: FFT

FFTCOMP Function

Computes the discrete Fourier transform of a real or complex sequence. Using keywords, a real-to-complex transform or a two-dimensional complex Fourier transform can be computed.

Usage

result = FFTCOMP(*a*)

Input Parameters

a — An array containing the periodic sequence.

Returned Value

result — The transformed sequence. If *a* is one-dimensional, the type of *a* determines whether the real or complex transform is computed. If *a* is two-dimensional, the complex transform is always computed.

Keywords

Backward — If present and nonzero, the backward transform is computed. See the *Discussion* section for more details on this option.

Complex — If present and nonzero, the complex transform is computed. If *a* is complex, this keyword is not required to ensure that a complex transform is computed. If *a* is real, it is promoted to complex internally.

Double — If present and nonzero, double precision is used.

Init_Params — An array containing parameters used when computing a one-dimensional FFT. If FFTCOMP is used repeatedly with arrays of the same length and data type, it is more efficient to compute these parameters only once with a call to function FFTINIT.

Discussion

The default action of the function FFTCOMP is to compute the FFT of an array *A*, with the type of FFT performed dependent upon the data type of the input array *A*. (If *A* is a one-dimensional real array, the real FFT is computed; if *A* is a one-dimensional complex array, the complex FFT is computed; and if *A* is a two-dimensional real or complex array, the complex FFT is computed.) If the complex FFT of a one-dimensional real array is desired, keyword *Complex* should be specified. The remainder of this section is divided into separate discussions of real and complex FFTs.

Case 1: One-dimensional Real FFT

If *A* is one-dimensional and real, the function FFTCOMP computes the discrete Fourier transform of a real array of length
n = N_ELEMENTS (*A*). The method used is a variant of the Cooley-Tukey algorithm, which is most efficient when *n* is a product of small prime factors. If *n* satisfies this condition, then the computational effort is proportional to *n*log*n*.

By default, FFTCOMP computes the forward transform. If *n* is even, the forward transform is as follows:

$$q_{2m-1} = \sum_{k=0}^{n-1} p_k \cos\frac{2\pi km}{n}$$

$$q_{2m-2} = -\sum_{k=0}^{n-1} p_k \sin\frac{2\pi km}{n}$$

$$q_0 = \sum_{k=0}^{n-1} p_k$$

If *n* is odd, q_m is defined as above for *m* from 1 to *n* – 1.

If the keyword *Backward* is specified, the backward transform is computed. If *n* is even, the backward transform is as follows:

$$q_m = p_0 + (-1)^m p_{n-1} + 2\sum_{k=0}^{\frac{n}{2}-1} p_{2k+1}\left(\cos\frac{2\pi km}{n}\right) - 2\sum_{k=0}^{\frac{n}{2}-1} p_{2k+2}\left(\sin\frac{2\pi km}{n}\right)$$

If *n* is odd, the following is true:

$$q_m = p_0 + 2\sum_{k=0}^{\frac{n-3}{2}} p_{2k+1}\left(\cos\frac{2\pi km}{n}\right) - 2\sum_{k=0}^{\frac{n-3}{2}} p_{2k+2}\left(\sin\frac{2\pi km}{n}\right)$$

The backward Fourier transform is the non-normalized inverse of the forward Fourier transform.

The FFTCOMP function is based on the real FFT in FFTPACK, which was developed by Paul Swarztrauber at the National Center for Atmospheric Research.

Case 2: One-dimensional Complex FFT

If *A* is one-dimensional and complex, function FFTCOMP computes the discrete Fourier transform of a complex array of size
n = N_ELEMENTS (*A*). The method used is a variant of the Cooley-Tukey algorithm, which is most efficient when *n* is a product of small prime factors. If *n* satisfies this condition, the computational effort is proportional to *n*log*n*.

By default, FFTCOMP computes the forward transform as in the following equation.

$$q_k = \sum_{m=0}^{n-1} p_m e^{(-2\pi jmk)/n} .$$

Note, the Fourier transform can be inverted as follows:

$$p_m = \frac{1}{n}\sum_{k=0}^{n-1} q_j e^{2\pi jk(m/n)} .$$

This formula reveals the fact that, after properly normalizing the Fourier coefficients, you have coefficients for a trigonometric polynomial to interpolate the data.

If the keyword *Backward* is used, the following computation is performed:

$$q_k = \sum_{m=0}^{n-1} p_m e^{2\pi jm(k/n)} .$$

Furthermore, the relation between the forward and backward transforms is that they are non-normalized inverses of each other. In other words, the following code fragment begins with an array p and concludes with an array $p_2 = np$:

```
q  = FFTCOMP(p)
p2 = FFTCOMP(q, /Backward)
```

Case 3: Two-dimensional FFT

If *A* is two-dimensional and real or complex, function FFTCOMP computes the discrete Fourier transform of a two-dimensional complex array of size *n* x *m* where
n = N_ELEMENTS (*A* (*, 0)) and
n = N_ELEMENTS (*A* (0, *)). The method used is a variant of the Cooley-Tukey algorithm, which is most efficient when both *n* and *m* are a product of small prime factors. If *n* and *m* satisfy this condition, then the computational effort is proportional to *nm*log*nm*.

By default, given a two-dimensional array, FFTCOMP computes the forward transform as in the following equation:

$$q_{kl} = \sum_{s=0}^{n-1} \sum_{t=0}^{m-1} p_{st} e^{-2\pi jks/n} e^{-2\pi jlt/m}$$

Note, the Fourier transform can be inverted as follows:

$$p_{kl} = \frac{1}{nm} \sum_{s=0}^{n-1} \sum_{t=0}^{m-1} q_{st} e^{2\pi jks/n} e^{2\pi jlt/m}$$

This formula reveals the fact that, after properly normalizing the Fourier coefficients, you have the coefficients for a trigonometric polynomial to interpolate the data.

If keyword *Backward* is used, the following computation is performed:

$$p_{kl} = \sum_{s=0}^{n-1}\sum_{t=0}^{m-1} q_{st}e^{2\pi jks/n}e^{2\pi jlt/m}$$

Example

In this example, a pure cosine wave is used as a data array, and its Fourier series is recovered. The Fourier series is an array with all components zero except at the appropriate frequency where it has an $n/2$.

```
n = 7
    ; Fill up the data array with a pure cosine wave.
p = COS(FINDGEN(n) * 2 * !Pi/n)
PM, p
      1.00000
      0.623490
    -0.222521
    -0.900969
    -0.222521
      0.623490
q = FFTCOMP(p)
    ; Compute the FFT.
PM, q, Format = '(f8.3)'
      0.000
      3.500
      0.000
    -0.000
    -0.000
      0.000
    -0.000
```

See Also

DCMPLXFFT, FFTINIT

In the *PV-WAVE Reference*:
FFT

FFTINIT Function

Computes the parameters for a one-dimensional FFT to be used in function FFT-COMP with keyword *Init_Params*.

Usage

result = FFTINIT(*n*)

Input Parameters

n — Length of the sequence to be transformed.

Returned Value

result — A one-dimensional array of length 2*n* + 15.

NOTE The resulting array can be used in the function FFTCOMP, along with the optional keyword *Init_Params*.

Keywords

Complex — If present and nonzero, the parameters for a complex transform are computed.

Double — If present and nonzero, double precision is used and the returned array is double precision. This keyword does not have an effect if the initialization is being computed for a complex FFT.

Discussion

FFTINIT should be used when many calls are to be made to function FFTCOMP without changing the data type of the array and the length of the sequence. The default action of FFTINIT is to compute the parameters necessary for a real FFT. If parameters for a complex FFT are needed, keyword *Complex* should be specified.

The FFTINIT function is based on routines in FFTPACK, which was developed by Paul Swarztrauber at the National Center for Atmospheric Research.

See Also

FFTCOMP

FILTDOWN Function

Decimation filter realization.

Usage

result = FILTDOWN([*h*,] *x*, *m*)

Input Parameters

h — (optional) An FIR filter. (Default: $H(z) = 1$)

x — A one-dimensional array containing the signal.

m — (scalar) The decimation factor. The parameter *m* must be less than or equal to the length of *x*.

Returned Value

result — A one-dimensional array in double precision containing the filtered and down-sampled input sequence.

Keywords

Full — Directs FILTDOWN to call FIRFILT with the *Full* keyword. If present and nonzero, the entire result of the convolution of the input signal is returned, including the edge effects which are normally cut off.

Discussion

FILTDOWN is designed to only work with FIR filters.

FILTDOWN realizes the multirate signal processing operation of filtering followed by down-sampling (decimation) for finite impulse response (FIR) filters, $H(z)$, as shown in *Figure 2-2*.

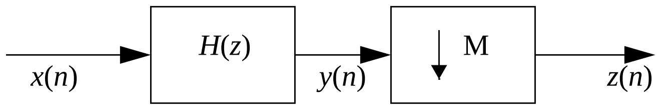


Figure 2-2 The FILTDOWN operation.

The down-sampling operation illustrated in *Figure 2-2* is given by the equation

$$z(n) = y(Mn)\ .$$

Example 1

Suppose *x* is an input signal of length 100. FILTDOWN is used to down-sample *x*, so that only every third point in the signal array is returned.

```
x = RANDOM(100)

PRINT, N_ELEMENTS(x)
   100

newx = FILTDOWN(x, 3)

PRINT, N_ELEMENTS(newx)
   34
```

Example 2

In this example, an FIR filter is applied to a signal, and only every third point is returned.

```
t = FINDGEN(100)/99

x = SIN(2*!Pi*t) + .4*RANDOM(100)

result = FILTDOWN(FILTSTR(FIRWIN(5), [1]), x, 3)
```

See Also

FILTDOWNDESIGN, FILTUP

For Additional Information

Crochiere and Rabiner, 1975, 1976.

Proakis and Manolakis, 1988.

Vaidyanathan, 1993.

FILTDOWNDESIGN Function

Designs a filter to be used with FILTDOWN.

Usage

result = FILTDOWNDESIGN(*m*, *nx*)

Input Parameters

m — (scalar) The decimation factor indicating the increment between data values to be returned. This is the input parameter *m* that is passed into FILTDOWN with the returned filter.

nx — The number of samples of the signal to be used with FILTDOWN.

Returned Value

result — A filter structure to be used with FILTDOWN.

Keywords

None.

Discussion

FILTDOWNDESIGN uses FIRDESIGN to approximate an anti-aliasing filter with normalized cutoff frequency of 1/*m*. A Hamming window is used in the filter approximation.The order of this decimation filter is 1/5 the length of the signal, *nx*, but never smaller than 16 and never greater than 128.

You may want to use the REMEZ function directly to create your own anti-aliasing filter.

Example

In this example every other sample is removed from *x*.

```
t = FINDGEN(100)/99

x = SIN(2*!Pi*t) + .4*RANDOM(100)
    ; Create a signal.

m = 2
    ; Removes every other point of the filtered signal.

nx = N_ELEMENTS(x)

h = FILTDOWNDESIGN(m, nx)
```

; Create an anti-aliasing filter.

```
newx = FILTDOWN(h, x, m)
```

; Filter and down-sample.

```
resp = ABS(FREQRESP_Z(h, Outfreq = f))
PLOT, f, resp, XTitle = '1/m', Title = $
    'Filter Response for Various m'
```

; Plot the response of this filter (see *Figure 2-3*).

```
OPLOT, f, ABS(FREQRESP_Z(FILTDOWNDESIGN(3, nx))),$
    Linestyle = 1
OPLOT, f, ABS(FREQRESP_Z(FILTDOWNDESIGN(4, nx))),$
    Linestyle = 2
OPLOT, f, ABS(FREQRESP_Z(FILTDOWNDESIGN(5, nx))),$
    Linestyle = 3
OPLOT, f, ABS(FREQRESP_Z(FILTDOWNDESIGN(6, nx))),$
    Linestyle = 4
```

; Plot the response of the filter for greater values of *m* to show
; how the cutoff frequency decreases as *m* increases (*Figure 2-3*).

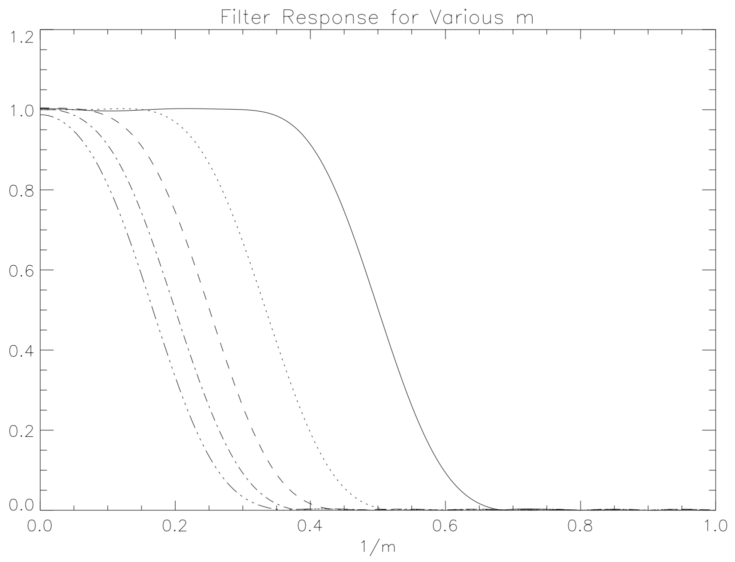


Figure 2-3 Decreasing cutoff frequency for increasing values of *m*, the decimation factor.

See Also

FILTDOWN, FIRFILT, REMEZ

FILTER Function

Applies an IIR or FIR filter to a sequence.

Usage

result = FILTER(*h*, *x*)

Input Parameters

h — A valid FIR or IIR filter structure.

x — A one-dimensional array to be filtered.

Returned Value

result — A one-dimensional array containing the filtered values of *x*.

Keywords

None.

Discussion

FILTER simplifies access to the filtering methods available in the PV-WAVE:Signal Processing Toolkit. FILTER determines whether the filter structure being used, *h*, is an FIR or an IIR filter type. It then calls the specific filtering routine appropriate for the filter structure used (FIRFILT or IIRFILT).

The particular FIR or IIR filtering routine called by FILTER uses all the default settings for the keyword parameters of that routine. For greater control of the filtering method used, however, it is recommended that you use the appropriate filter function directly (see FIRFILT, IIRFILT).

Example

The following example shows a typical application of FILTER.

```
!P.Multi = [0, 1, 2]

t = FINDGEN(1024)

s1 = SIN(0.6*t)

s2 = SIN(1.2*t)
```

```
s3 = SIN(1.9*t)
```

; Generate three bandpass signals.

```
x = s1 + s2 + s3
```

; Combine the three signals.

```
f = FINDGEN(512)/511
```

; Generate abscissa values for the normalized frequency.

```
PLOT, f, (ABS(FFTCOMP(x, /Complex)))(0:512), $
   Title = 'Original', XStyle = 1
```

; Plot magnitude frequency response of combined
; signal (*Figure 2-4* (a)).

```
h = FIRDESIGN(101, 0.3, 0.5, /Bandpass)
```

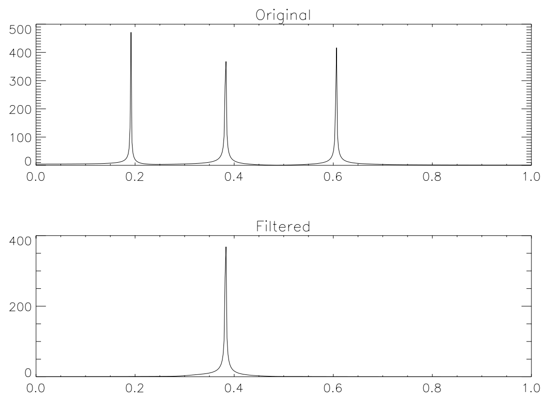
; Approximate a bandpass filter to isolate the first signal.

```
y = FIRFILT(h, x)
```

; Apply the filter to the signal.

```
PLOT, f, (ABS(FFTCOMP(y, /Complex)))(0:512), $
   Title = 'Filtered', XStyle = 1
```

; Plot magnitude frequency response of filtered
; signal (*Figure 2-4* (b)).



(a)

(b)

Figure 2-4 (a) Plot of the magnitude frequency response of the combined signal. (b) Plot of the magnitude frequency response of the filtered signal.

See Also

FILTSTR, FIRFILT, IIRFILT

FILTSTR Function

Constructs a valid filter data structure.

Usage

h = FILTSTR(*b*, *a*)

Input Parameters

b — A one-dimensional array or scalar value representing the numerator of the filter.

a — A one-dimensional array or scalar value representing the denominator of the filter.

Returned Value

h — A structure containing the filter.

Keywords

Name — A scalar string containing a name for the filter.

Discussion

For the numerator polynomial $B(z)$ such that

$$B(z) = b_0 + b_1z^{-1} + b_2z^{-2} + \ldots$$

and the dominator polynomial $A(z)$ such that

$$A(z) = a_0 + a_1z^{-1} + a_2z^{-2} + \ldots,$$

FILTSTR constructs a filter data structure $H(z)$ such that

$$H(z) = B(z)/A(z).$$

Example

FILTSTR is used to produce a filter $H(z)$ such that

$$H(z) = \frac{(1 + z^{-3})}{(5 + z^{-1} + 2z^{-2})} .$$

```
h = FILTSTR([1, 0, 0, 1],[5, 1, 2])
```

See Also

PARSEFILT

FILTUP Function

Interpolation filter realization.

Usage

result = FILTUP([*h*,] *x*, *m*)

Input Parameters

h — (optional) An FIR filter. (Default: *H*(*z*) = 1)

x — A one-dimensional array containing a signal.

m — (scalar) The interpolation factor.

Returned Value

result — A one-dimensional array in double precision containing the filtered and up-sampled input sequence.

Keywords

Full — Directs FILTDOWN to call FIRFILT with the *Full* keyword. If present and nonzero, the entire result of the convolution of the input signal is returned, including the edge effects which are normally cut off.

Discussion

FILTUP is designed for use only with FIR filters.

FILTUP realizes the multirate signal processing operation of up-sampling followed by filtering (interpolation) for finite impulse response (FIR) filters, *H*(*z*) as shown in *Figure 2-5*.

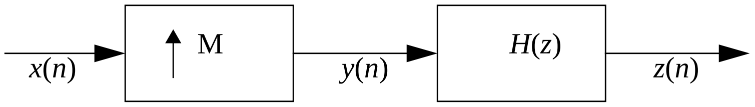


Figure 2-5 The FILTUP operation.

The up-sampling shown in *Figure 2-5* is given by

$$y(n) = \begin{cases} x(n), & n = 0, \pm M, \pm 2M, \ldots \\ 0, & \text{otherwise} \end{cases}$$

Example

In this example, a signal is created and then up-sampled to make a signal with four times as many samples.

```
x = SIN(0.01*INDGEN(1000))
    ; Define a signal.
m = 4
    ; The interpolation factor of 4 means that three new samples
    ; will be inserted between each sample of the original signal.
h = FILTUPDESIGN(m, 2)
    ; Create an interpolation filter.
newx = FILTUP(h, x, m)
    ; Interpolate the data, using the interpolation filter. This new
    ; signal is approximately 4 times the length of the original signal.
```

See Also

FILTDOWN, FILTUPDESIGN, FIRFILT

FILTUPDESIGN Function

Designs a filter to be used with FILTUP.

Usage

result = FILTUPDESIGN(*m*, *l* [, *alpha*])

Input Parameters

m — (scalar) The interpolation factor. This should be the same value as that being passed into FILTUP.

l — An integer used to determine the order of the filter. The filter order is given by $n = 2lm$.

alpha — (optional) A scalar value used to scale the filter cutoff frequency. The normalized cutoff frequency is given by *alpha* divided by *m*. (Default: *alpha* = 1)

Returned Value

result — A filter structure to be used with FILTUP.

Keywords

None.

Discussion

FILTUPDESIGN generates an optimal linear phase interpolator filter. The filter generated has the property that every *m*-th point of the interpolated signal is equal to a corresponding point in the original sequence.

The filter design technique implemented in FILTUPDESIGN is based on a paper by Oetken, Parks, and Schussler, 1975.

Example

In this example, a signal is interpolated to get a signal four times the length of the original.

```
m = 4
```

```
nx = 100
```

```
freq = 0.9

x = SIN(!Pi*(freq)*FINDGEN(nx))
      ; Create a signal.

alpha = 1.0

l = 4
      ; This will cause the resulting filter to be of order 32.

h = FILTUPDESIGN(m, l, alpha)
      ; Create the interpolation filter.

result = FILTUP(h, x, m)
      ; Interpolate and filter in one step.

newx = FILTDOWN(result, m)
      ; To check the interpolation filter, decimate the interpolated, filtered
      ; data. The original data set is returned, to within a close epsilon.

PRINT, TOTAL(ABS(newx - x))
    2.5938099e-14
          ; The difference between the original and the decimated interpolation
          ; filter data.
```

See Also

FILTUP

For Additional Information

Oetken, Parks, and Schussler, 1975.

FIRDESIGN Function

Designs windowed, finite impulse response (FIR) digital filters including lowpass, highpass, bandpass, and bandstop filters.

Usage

result = FIRDESIGN(*w*, *f1* [, *f2*])

result = FIRDESIGN(*m*, *f1* [, *f2*])

Input Parameters

w — A one-dimensional array containing a window sequence.

m — The filter length.

f1 — The frequency band edge for lowpass and highpass filters, or the lower frequency band edge for bandpass and bandstop filters.

f2 — The upper frequency band edge for bandpass and bandstop filters.

Returned Value

result — A filter structure containing an FIR filter.

Keywords

Bandpass — If present and nonzero, designs a bandpass filter.

Bandstop — If present and nonzero, designs a bandstop filter.

Highpass — If present and nonzero, designs a highpass filter.

Lowpass — If present and nonzero, designs a lowpass filter.

Discussion

FIRDESIGN designs linear phase FIR filters to approximate the ideal lowpass, highpass, bandpass, and bandstop filters shown in *Figure 2-6*.

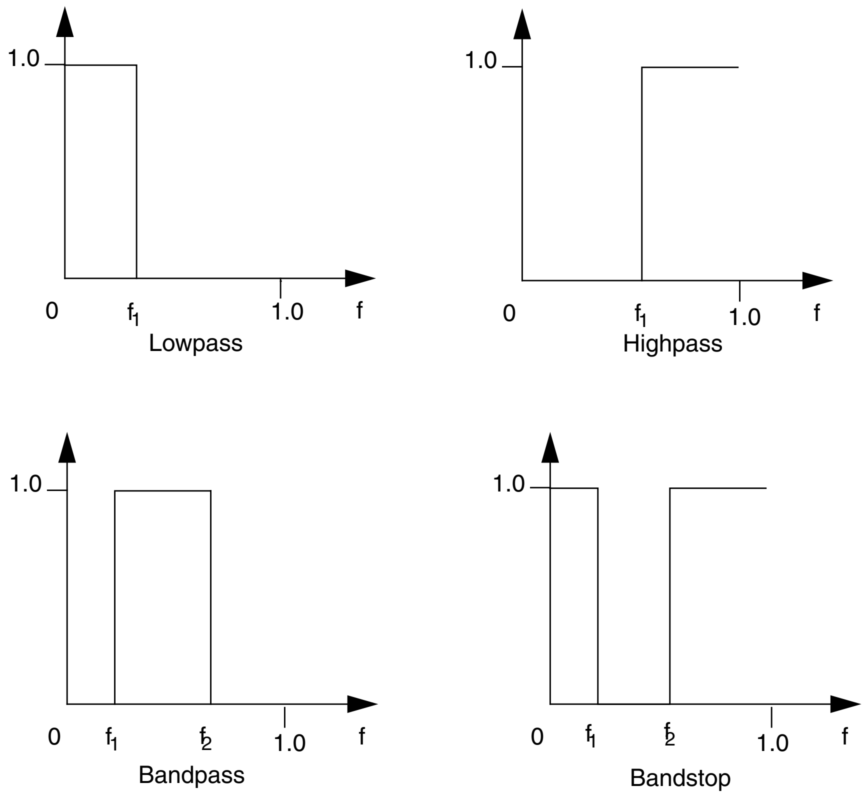


Figure 2-6 Ideal lowpass, highpass, bandpass, and bandstop filters can be approximated with linear phase FIR filters designed with FIRDESIGN.

The basic approximation technique begins with one of the four ideal frequency responses

$$|H_{\text{Ideal}}(e^{j\pi f})| = \begin{cases} 1, & f \in B \\ 0, & f \notin B \end{cases}$$

for an appropriate frequency set *B*. The impulse response

$$_{\text{Ideal}}(k) = \frac{1}{2\pi}\int_{-\pi}^{\pi} H_{\text{Ideal}}(e^{j\theta})e^{jk\theta}d\theta,\ (\theta = \pi f)$$

is then determined and multiplied by a window sequence $w(k)$, giving the result

$$b(k) = w(k)h_{Ideal}(k).$$

The values of $b(k)$ are returned in the filter structure as

$$H(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2} + \dots + b_m z^{-m}}{1}.$$

If the first parameter in the calling sequence is an array of length *m*, an FIR filter of length *m* is designed using this array as the window sequence. If the first parameter is an integer *m*, then by default an FIR filter of length *m* is designed using a Hamming window.

Example

In this example, FIRDESIGN is used to approximate an ideal bandpass filter with band edges 0.4 and 0.6 using a Blackman window.

```
w = FIRWIN(101, /Blackman)
   ; Compute a Blackman window sequence of length 101.
h = FIRDESIGN(w, 0.4, 0.6, /Bandpass)
   ; Approximate an ideal bandpass filter with band edges 0.4
   ; and 0.6 using a Blackman window.
hf = FREQRESP_Z(h, Outfreq = f)
PLOT, f, ABS(hf), Title = 'Magnitude', XTitle = 'Frequency'
   ; Plot the magnitude of frequency response (Figure 2-7).
```

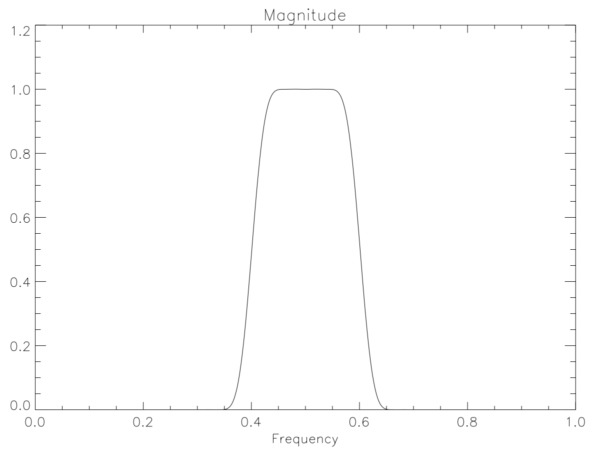


Figure 2-7 Plot of the magnitude of frequency response of an ideal bandpass filter.

See Also

FIRFILT, FIRLS, FIRWIN

For Additional Information

Mitra and Kaiser, 1993, section 4.4.

Oppenheim and Schafer, 1989, section 7.4.

FIRFILT Function

Applies a finite impulse response (FIR) filter to a sequence.

Usage

result = FIRFILT(*h*, *x*)

Input Parameters

h — A filter structure containing an FIR filter.

x — A one-dimensional array containing the data signal to be filtered.

Returned Value

result — A one-dimensional array of the same length as *x* containing the filtered data.

Keywords

Full — If present and nonzero, the entire result of the convolution of the signal with the filter is returned.

NOTE *Full* is not valid if keyword *Periodic* is also set.

Offset — The result is shifted by the offset after the filter operation. (Default: N_ELEMENTS(*h.b*)/2)

Periodic — If present and nonzero, *x* is treated as periodic.

Discussion

For a given filter structure of the form

$$H(z) = B(z)/A(z) \quad \text{with} \quad A(z) = 1,$$

FIRFILT computes the output sequence $y(k)$ from the input sequence $x(k)$ as

$$y(k) = \sum_{l} b_{k-l} x(k),$$

where b_k are the coefficients of $B(z)$.

Function CONVOL1D is used to carry out this computation. In order to reduce end effects, a portion of the result of the convolution operation is removed from the beginning and the end of the filtered signal before being returned. Use keyword *Offset* to adjust how the elements are removed.

The length of *result* is the same as the input parameter *x* by default. To return the entire result of the convolution operation, use keyword *Full*.

Example

This example illustrates a typical application of FIRFILT.

```
!P.Multi = [0, 1, 2]

t = FINDGEN(1024)
s1 = SIN(0.6*t)
s2 = SIN(1.2*t)
s3 = SIN(1.9*t)
   ; Generate three bandpass signals.

x = s1 + s2 + s3
   ; Combine the three signals.

f = FINDGEN(512)/511
   ; Generate abscissa values for the normalized frequency.

PLOT, f, (ABS(FFTCOMP(x, /Complex)))(0:512), $
   Title = 'Original', XStyle = 1
      ; Plot the magnitude frequency response of the combined
      ; signal (Figure 2-8 (a)).

h = FIRDESIGN(101, 0.1, 0.3, /Bandpass)
   ; Approximate a bandpass filter to isolate the first signal.

y = FIRFILT(h, x)
```

; Apply the filter to the signal.

```
PLOT, f, (ABS(FFTCOMP(y,/Complex)))(0:512), $
   Title = 'Filtered', XStyle = 1
```

; Plot the magnitude frequency response of the filtered
; signal (*Figure 2-8* (b)).

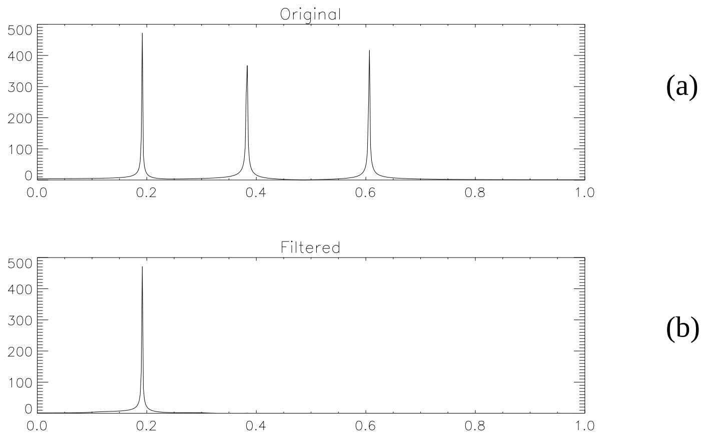


Figure 2-8 (a) Plot of the magnitude frequency response of the combined signal. (b) Plot of the magnitude frequency response of the filtered signal.

See Also

FIRDESIGN, FIRLS, FIRWIN

For Additional Information

Oppenheim and Schafer, 1989.

Proakis and Manolakis, 1992.

FIRLS Function

Approximates a deterministic frequency-domain or statistical time-domain, least-squares, linear-phase finite impulse response (FIR) filter.

Usage

result = FIRLS(*n*, *f*, *amplitude*)

result = FIRLS(*n*, *rss*, *rnn*, /*Wiener*)

Input Parameters

n — The length of the desired FIR filter.

f — An array of frequency points between 0.0 and 1.0.

amplitude — The desired filter amplitude response at the frequency points specified by *f*. Amplitude values may be positive or negative.

rss — The autocorrelation sequence of the signal.

rnn — The autocorrelation sequence of the noise.

Returned Value

result — A filter structure containing the FIR filter approximation.

Keywords

Freqsample — If present and nonzero, approximates a linear phase FIR filter using frequency domain least-squares techniques.

Interpfactor — An integer value specifying a factor by which the frequency samples are interpolated. The total number of interpolation points is given by *Interpfactor*n*. (Default: 4)

Oddsymm — If present and nonzero, the FIR filter approximated using deterministic frequency domain techniques will have odd symmetry.

Wiener — If present and nonzero, an odd length, even symmetric, linear phase FIR filter is approximated using statistical time domain least-squares (Wiener) techniques.

Discussion

When the keyword *Freqsample* is specified, FIRLS produces linear phase filters that approximate a desired amplitude frequency response. Four types of linear phase FIR filters can be approximated. They are:

Type 1 filters have odd length $n = 2m + 1$, even symmetry, and are given by

$$G(z) = g(0) + \sum_{n=1}^{m} g(n)(z^n + z^{-n}) \quad .$$

Type 2 filters have even length $n = 2m$, even symmetry, and are given by

$$G(z) = \sum_{n=0}^{m-1} g(n)\left(z^{n+\frac{1}{2}} + z^{-n-\frac{1}{2}}\right) \; .$$

Type 3 filters have odd length $n = 2m + 1$, odd symmetry, and are given by

$$G(z) = \sum_{n=1}^{m} g(n)(z^n - z^{-n}), \quad \mathrm{g}(0) = 0 \quad .$$

Type 4 filters have even length $n = 2m$, odd symmetry, and are given by

$$G(z) = \sum_{n=0}^{m-1} g(n)\left(z^{n+\frac{1}{2}} - z^{-n-\frac{1}{2}}\right) \; .$$

The different types of filters are chosen by selecting the length n to be odd or even and by setting the keyword *Oddsymm*.

FIRLS requires the specification of a selection of distinct frequency points between 0 and 1 in increasing order

$$f = [f_0, f_1, \ldots , f_L]$$

and the desired frequency amplitude responses

$$\text{amplitude} = [G(z_0), G(z_1), \ldots , G(z_L)]$$

at these frequency points where

$$_k = e^{j\pi f_k}.$$

The algorithm used to design a Type 1 linear phase filter takes the specified frequency and amplitude points and then forms a matrix *A*, with the *k*, *n* element given by

$$^{jn\pi f_k} + e^{-jn\pi f_k} = 2\cos(n\pi f_k),$$

and an array *b* with the elements of the desired frequency amplitude responses. The equation $Ax = b$ is then solved for the array *x*, which contains the filter coefficients $g(n)$, $n = 0, 1, 2, ..., m$. The other three types of filters are obtained in an identical manner, except the elements of the matrix *A* are changed appropriately.

The solution to the matrix equation depends upon the number of frequency sample points provided. If a filter of the length *n* has *L* frequency points specified and the filter has $m = (n – 1)/2$ distinct coefficients for *n* odd (or $m = n/2$ for *n* even), then the different solutions are enumerated as follows.

For the case when *n* is odd and the desired filter has even symmetry, FIRLS returns the following solutions:

- If $L – 1 > m$, *result* is a least-squares FIR filter obtained using the specified frequency and amplitude points.
- If $L – 1 = m$, *result* is the solution to the trigonometric polynomial interpolation problem.
- If $L – 1 < m$, the frequency samples are linearly interpolated onto a uniform grid of 4**n* points, and *result* is a least-squares FIR filter based on the interpolated frequency points.

For all other cases, FIRLS returns the following solutions:

- If $L > m$, *result* is a least-squares FIR filter obtained using the specified frequency and amplitude points.
- If $L = m$, *result* is the solution to the trigonometric polynomial interpolation problem.
- If $L < m$, the frequency samples are linearly interpolated onto a uniform grid of 4**n* points, and *result* is a least-squares FIR filter based on the interpolated frequency points.
- If *Interpfactor* is specified, the default solutions are overridden and the frequency samples are linearly interpolated onto a uniform grid of *Interpfactor***n* points. The least squares solution based on the interpolated frequency points determines the filter coefficients.

The coefficients of the four types of linear phase FIR filters are related to the coefficients of the returned filter

$$H(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2} + \dots + b_m z^{-m}}{1}$$

as $H(z) = z^{-n}G(z)$, where $m = n - 1$.

Further details of the equations used in the solution can be found in Parks and Burrus, 1987, sections 7.4 and 7.5.

NOTE FIR filters do not have an approximation problem equivalent to the time-domain least squares problem for IIR filters (see the IIRLS *Discussion* section). However, a time-domain statistical least-squares approximation problem does exist for FIR filters which are solved by FIRLS.

When *Wiener* is specified, FIRLS designs an optimal Type 1 linear phase FIR filter for estimating the signal $s(k)$ from the noise corrupted observation

$$x(k) = s(k) + n(k).$$

It is assumed that the signal and noise are stationary, have zero mean and have the known autocorrelation sequences

$$r_{ss}(k) = E[s(l)s(l + k)]\ ,\ k = 0, \dots, m$$

and

$$r_{nn}(k) = E[n(l)n(l + k)]\ ,\ k = 0, \dots, m$$

where E is the mathematical expectation operator.

The coefficients of the FIR filter

$$H(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2} + \dots + b_m z^{-m}}{1}$$

are chosen to minimize the error

$$E\left[s(k) - \sum_i b(i)u(k - i)\right]^2$$

subject to the constraint that the number of coefficients is odd and the filter has linear phase.

The solution to this problem is obtained by solving the following linear equation

$$[R_{ss}(2L) + R_{nn}(2L)]\begin{bmatrix} b(0) \\ | \\ b(L) \\ | \\ b(2L) \end{bmatrix} = R_{ss}(2L)\begin{bmatrix} 0 \\ | \\ 0 \\ 1 \\ 0 \\ | \\ 0 \end{bmatrix} \begin{matrix} \left.\begin{matrix} \\ \\ \\ \end{matrix}\right\} L \text{ zeros} \\ \\ \left.\begin{matrix} \\ \\ \\ \end{matrix}\right\} L \text{ zeros} \end{matrix}$$

where the filter length $n = 2L + 1$ and the matrices R_{ss} and R_{nn} are Toeplitz, defined by

$$R_{xx}(m) = \begin{bmatrix} r_{xx}(0) & r_{xx}(1) & \ddots & r_{xx}(m) \\ r_{xx}(1) & \ddots & \ddots & \\ & \ddots & \ddots & r_{xx}(1) \\ r_{xx}(m) & & r_{xx}(1) & r_{xx}(0) \end{bmatrix} .$$

NOTE When *Wiener* is specified, the two covariance sequences r_{ss} and r_{nn} must be positive definite.

NOTE When *Freqsample* is specified and an exact interpolation of the frequency samples is required, then:

- When n is odd and *Oddsymm* is present, f cannot equal 0.0 or 1.0.
- When n is even and the desired filter has even symmetry (*Oddsymm* is not present), f cannot equal 1.0.
- When n is even and *Oddsymm* is present, f cannot equal 0.0.

CAUTION Take care when choosing the frequency points and amplitude values. If there are sharp transitions in the amplitude response, better solutions can be obtained by specifying a transition region. FIRLS is purposefully set up to allow you to specify the transition bands to suit your needs. This feature provides considerable design flexibility.

Example 1

In this example, a Type 1 FIR filter is designed to interpolate a given set of frequency amplitude points.

```
h1 = FIRDESIGN(11,0.5,/Lowpass)
   ; Design an FIR filter.
PARSEFILT, h1, name, numer1, denom1
PM, numer1, Title = 'Original FIR Filter Coefficients'
   Original FIR Filter Coefficients
        0.0000000
       -1.8611500e-18
       -0.036657787
        1.2756512e-17
        0.28791400
        0.50000000
        0.28791400
        1.2756512e-17
       -0.036657787
       -1.8611500e-18
        0.0000000
f = FINDGEN(5)/5.0
desired_amplitude = ABS(FREQRESP_Z(h1, Infreq = f))
   ; Compute the frequency response of filter at several points where
   ; the amplitude response is known to be positive.
h2 = FIRLS(11, f, desired_amplitude, /Freqsample)
   ; Design an FIR filter to interpolate the desired amplitude values.
PARSEFILT, h2, name, numer2, denom2
PM, numer2, Title = 'Interpolating FIR Filter Coefficients'
   Interpolating FIR Filter Coefficients
        6.2218717e-10
        0.0000000
       -0.036657792
        9.2945523e-11
        0.28791401
        0.50000002
        0.28791401
        9.2945523e-11
       -0.036657792
        0.0000000
        6.2218717e-10
```

Example 2

This example illustrates how to design a multiple bandpass filter using FIRLS and specified transition bands.

```
f = [0, .18, .2, .22, .38, .4, .42, .58, .6, .62,$
   .78, .8, .82, 1]
ampl = [0, 0, .5, 1, 1, .5, 0, 0, .5, 1, 1, .5, 0, 0]
```

; Specify desired frequency response with transition bands.

```
h = FIRLS(101, f, ampl, /Freqsample)
```

; Design a least-squares filter to approximate the desired response.

```
hf = FREQRESP_Z(h, Outfreq = f)
PLOT, f, ABS(hf), Title = 'Magnitude', $
   XTitle = 'Frequency'
```

; Plot the magnitude of the filter frequency response (*Figure 2-9*).

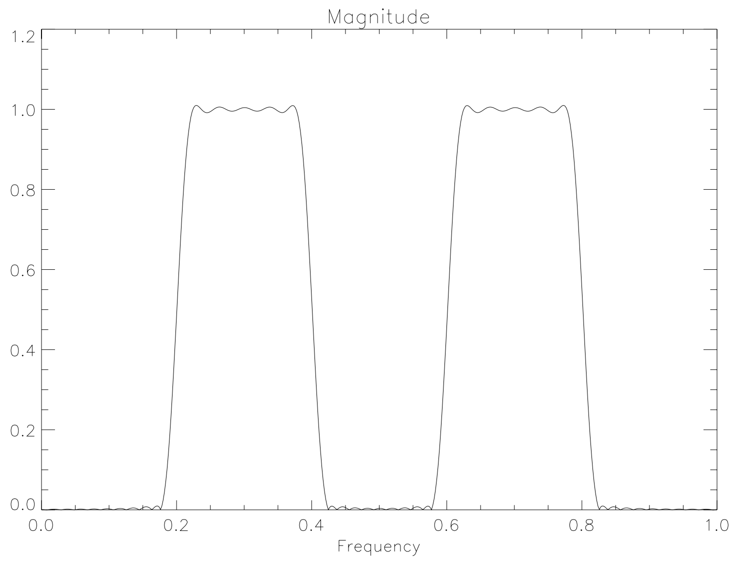


Figure 2-9 The magnitude plot of the multiple bandpass filter frequency response.

Example 3

This example illustrates the use of FIRLS with the keyword *Wiener* to design a statistical least-squares linear phase FIR filter.

```
rss = [1, .9, .9^2, .9^3, .9^4, .9^5, .9^6, .9^7, $
   .9^8]
```

; Covariance for first order autoregressive signal.

```
rnn = [.8, 0, 0, 0, 0, 0, 0, 0, 0]
```

; Covariance for white noise with variance 0.8.

```
h = FIRLS(9, rss, rnn, /Wiener)
```

; Compute the Wiener filter coefficients.

```
hf = FREQRESP_Z(h, Outfreq = f)
PLOT, f, ABS(hf), Title = 'Wiener Filter Magnitude', $
   XTitle = 'Frequency'
```

; Plot the magnitude of the frequency response (*Figure 2-10*).

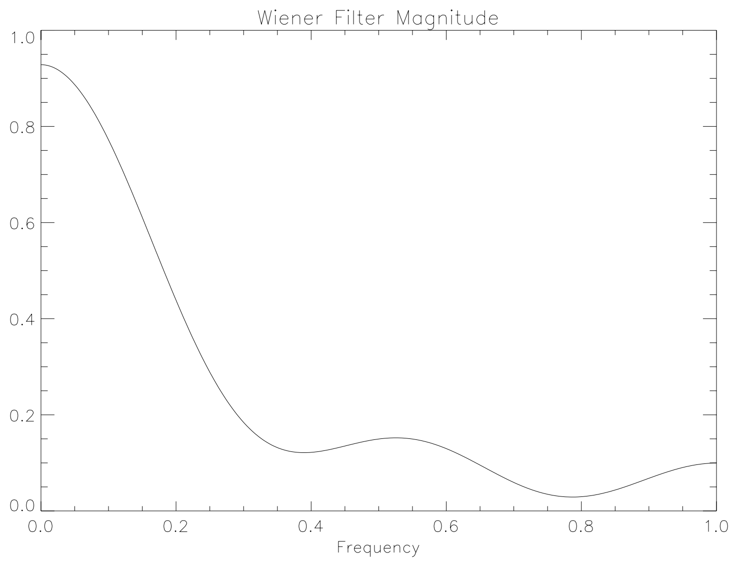


Figure 2-10 Plot of the magnitude of the Wiener filter frequency response.

See Also

FIRDESIGN, FIRFILT, FIRWIN

For Additional Information

Parks and Burrus, 1987.

Roberts and Mullis, 1987.

FIRWIN Function

Computes one of several different data windows: Blackman, Chebyshev, Hamming, Hanning, Kaiser, rectangular, or triangular.

Usage

result = FIRWIN(*n* [, *a*])

Input Parameters

n — The length of the window sequence.

a — (optional) A scalar float parameter used only when computing either the Kaiser or Chebyshev windows.

Returned Value

result — A one-dimensional array containing the requested window.

Keywords

Blackman — If set, a Blackman window is returned.

Chebyshev — If set, a Chebyshev window is returned. If *Chebyshev* is set, input parameter *a* (*a* is θ_0 in the Chebyshev window equation) is also required.

Hamming — If set, a Hamming window is returned.

Hanning — If set, a Hanning window is returned.

Kaiser — If set, a Kaiser window is returned. If *Kaiser* is set, input parameter *a* (*a**π is α in the Kaiser window equation) is also required.

Rectangular — If set, a rectangular window is returned. This is also known as a boxcar window.

Triangular — If set, a triangular window is returned.

Window_type — A scalar value (see the following table) used to specify a window type. This keyword may be used in place of the particular window keywords

Window	**Window_type**
1	Rectangular
2	Triangular
3	Hanning
4	Hamming
5	Kaiser
6	Blackman
7	Chebyshev

Discussion

A listing of the applicable time domain equations is given below, where ($w(n)$, $0 < n < N - 1$) for each of the windows.

Blackman window:

$$w(n) = 0.42 - 0.5\cos\frac{2\pi(n - \frac{N-1}{2})}{N-1} + 0.08\cos\frac{4\pi(n - \frac{N-1}{2})}{N-1}$$

Chebyshev window:

for $N = 0$,

$$w_0(k) = \begin{cases} 1, & \text{for k = 0} \\ 0, & \text{otherwise} \end{cases}$$

for $N = 1$,

$$w_1(k) = \begin{cases} x_0 - 1, & \text{for k = 0} \\ \frac{x_0}{2}, & \text{for |k| = 1} \\ 0, & \text{otherwise} \end{cases}$$

for $N > 1$,

$$w_N(k) = 2(x_0^2 - 1)w_{N-1}(k) +$$

$$(x_0^2)[w_{N-1}(k-1) + w_{N-1}(k+1)] - w_{N-2}(k)$$

where

$$x_0 = \frac{1}{\cos(\theta_0/2)}$$

and θ_0 is the FIRWIN input parameter *a*.

Hamming window:

$$w(n) = 0.54 - 0.46\cos\frac{2\pi(n - \frac{N-1}{2})}{N-1}$$

Hanning window:

$$w(n) = 0.5 - 0.5\cos\frac{2\pi(n - \frac{N-1}{2})}{N-1}$$

Kaiser window:

$$w(n) = \frac{I_0\left(\alpha\sqrt{1 - \left[2\frac{\left(n - \frac{N-1}{2}\right)}{N-1}\right]^2}\right)}{I_0(\alpha)} ,$$

where $I_0(x)$ is the zeroth order Bessel function of the first kind, and α is the FIR-WIN input parameter *a*.

Rectangular (or boxcar) window:

$w(n) = 1$, for all n.

Triangular window:

$$w(n) = 1 - \frac{2\left|n - \frac{N-1}{2}\right|}{N-1}, \text{ for all } n.$$

Example

This example illustrates how to generate each of the different window types for a window length of 31.

```
!P.Multi=[0,2,2]
```

```
n = 31

rect = FIRWIN(n, /Rectangular)

tri = FIRWIN(n, /Triangular)

hann = FIRWIN(n, /Hanning)

hamm = FIRWIN(n, /Hamming)

kaiser = FIRWIN(n, 0.5, /Kaiser)

black = FIRWIN(n, /Blackman)

cheby = FIRWIN(n, 0.15, /Chebyshev)

PLOT, rect, Linestyle = 2, XStyle = 1, $
   YRange = [0, 1.05], $
   Title = 'Rectangular and Triangular'
   ; See Figure 2-11 (a).

OPLOT, tri
   ; Plot the rectangular and triangular windows. See Figure 2-11 (a).

PLOT, hann, XStyle = 1, Linestyle = 2, $
   YRange = [0, 1.05], $
   Title = 'Hanning and Hamming'
      ; See Figure 2-11 (b).

OPLOT, hamm
   ; Plot the Hanning and Hamming windows. See Figure 2-11 (b).

PLOT, kaiser, XStyle = 1, Linestyle = 2, $
   YRange = [0, 1.05],$
   Title = 'Kaiser and Blackman'
      ; See Figure 2-11 (c).

OPLOT, black
   ; Plot the Kaiser and Blackman windows. See Figure 2-11 (c).

PLOT, cheby, XStyle = 1, YRange = [0, 1.05], $
   Title = 'Chebyshev'
      ; Plot a Chebyshev window. See Figure 2-11 (d).
```

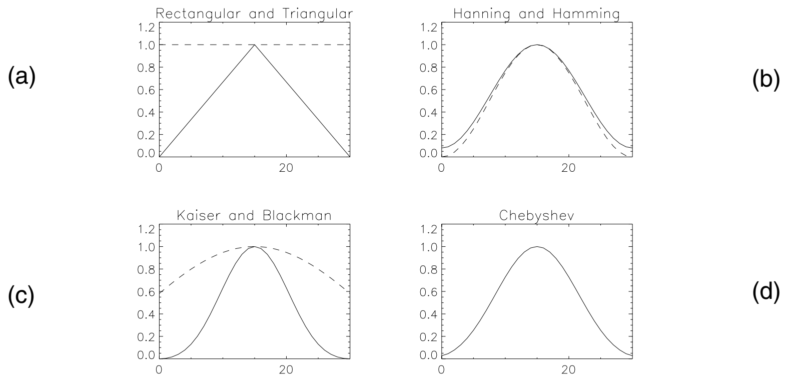


Figure 2-11 Plots of FIRWIN window types: (a) rectangular and triangular windows; (b) Hanning and Hamming windows; (c) Kaiser and Blackman windows; (d) Chebyshev window.

See Also

FIRDESIGN, FIRFILT, FIRLS

For Additional Information

Harris, 1978.

Roberts and Mullis, 1987.

FMIN Function

Finds the minimum point of a smooth function of a single variable *f* (*x*) using function evaluations and, optionally, through both function evaluations and first derivative evaluations.

Usage

result = FMIN(*f*, *a*, *b* [, *grad*])

Input Parameters

f — A scalar string specifying a user-supplied function to compute the value of the function to be minimized. Parameter *f* accepts the following parameter and returns the computed function value at this point:

> ***x*** — The point at which the function is to be evaluated.

a — The lower endpoint of the interval in which the minimum point of *f* is to be located.

b — The upper endpoint of the interval in which the minimum point of *f* is to be located.

grad — A scalar string specifying a user-supplied function to compute the first derivative of the function. Parameter *grad* accepts the following parameter and returns the computed derivative at this point:

> ***x*** — The point at which the derivative is to be evaluated.

Returned Value

result —The point at which a minimum value of *f* is found. If no value can be computed, then NaN (not a number) is returned.

Keywords

Double — If present and nonzero, double precision is used.

Err_Abs — The required absolute accuracy in the final value of *x*. On a normal return, there are points on either side of *x* within a distance *Err_Abs* at which *f* is no less than *f* at *x*. Keyword *Err_Abs* should not be used if the optional parameter *grad* is supplied. (Default: *Err_Abs* = 0.0001)

Err_Rel — The required relative accuracy in the final value of *x*. This is the first stopping criterion. On a normal return, the solution *x* is in an interval that contains a local minimum and is less than or equal to max (1.0, | *x* |) * *Err_Rel*. When the given *Err_Rel* is less than zero, $\varepsilon^{1/2}$ is used as *Err_Rel*, where ε is the machine precision. Keyword *Err_Rel* should only be used if the optional parameter *grad* is supplied. (Default: *Err_Rel* = $\varepsilon^{1/2}$)

FValue — The function value at point *x*. Keyword *FValue* should only be used if the optional parameter *grad* is supplied.

GValue — The derivative value at point *x*. Keyword *GValue* should only be used if the optional parameter *grad* is supplied.

Max_Evals — The maximum number of function evaluations allowed. (Default: *Max_Evals* = 1000)

Step — The order of magnitude estimate of the required change in *x*. Keyword *Step* should not be used if the optional parameter *grad* is supplied. (Default: *Step* = 1.0)

Tol_Grad — The derivative tolerance used to decide if the current point is a local minimum. This is the second stopping criterion. Parameter *x* is returned as a solution when *grad* is less than or equal to *Tol_Grad*. Keyword *Tol_Grad* should be nonnegative; otherwise, zero is used. Keyword *Tol_Grad* should only be used if the optional parameter *grad* is supplied. (Default: *Tol_Grad* = $\varepsilon^{1/2}$, where ε is the machine precision)

XGuess — The initial guess of the minimum point of *f*.
(Default: *XGuess* = (*a* + *b*) / 2)

Discussion

FMIN uses a safeguarded, quadratic interpolation method to find a minimum point of a univariate function. Both the code and the underlying algorithm are based on the subroutine ZXLSF written by M.J.D. Powell at the University of Cambridge.

The FMIN function finds the least value of a univariate function, *f*, which is specified by the function *f*. (Other required data are two points *A* and *B* that define an interval for finding a minimum point from an initial estimate of the solution, x_0, where x_0 = *XGuess*.) The algorithm begins the search by moving from x_0 to $x = x_0 + s$, where *s* = *Step* is an estimate of the required change in *x* and may be positive or negative. The first two function evaluations indicate the direction to the minimum point, and the search strides out along this direction until a bracket on a minimum point is found or until *x* reaches one of the endpoints *a* or *b*. During this stage, the step length increases by a factor of between 2 and 9 per function evaluation. The factor depends on the position of the minimum point that is predicted by quadratic interpolation of the three most recent function values.

When an interval containing a solution has been found, the three points are as follows:

x_1, x_2, x_3, with $x_1 < x_2 < x_3$, $f(x_1) \geq f(x_2)$, and $f(x_2) \geq f(x_3)$

The following rules should be considered when choosing the new x from these three points:

- the estimate of the minimum point that is given by quadratic interpolation of the three function values
- a tolerance parameter η, which depends on the closeness of $|f|$ to a quadratic
- whether x_2 is near the center of the range between x_1 and x_3 or is relatively close to an end of this range.

In outline, the new value of x is as near as possible to the predicted minimum point, subject to being at least ε from x_2 and subject to being in the longer interval between x_1 and x_2 or x_2 and x_3, when x_2 is particularly close to x_1 or x_3.

The algorithm is intended to provide fast convergence when f has a positive and continuous second derivative at the minimum and to avoid gross inefficiencies in pathological cases, such as the following:

$$f(x) = x + 1.001\,|x|$$

The algorithm can make ε large automatically in the pathological cases. In this case, it is usual for a new value of x to be at the midpoint of the longer interval that is adjacent to the least calculated function value. The midpoint strategy is used frequently when changes to f are dominated by computer rounding errors, which happens if you request an accuracy that is less than the square root of the machine precision. In such cases, the subroutine claims to have achieved the required accuracy if it decides that there is a local minimum point within distance δ of x, where $\delta = Err\_Abs$, even though the rounding errors in f may cause the existence of other local minimum points nearby. This difficulty is inevitable in minimization routines that use only function values, so high-precision arithmetic is recommended.

If parameter *grad* is supplied, then the FMIN function uses a descent method with either the secant method or cubic interpolation to find a minimum point of a univariate function. It starts with an initial guess and two endpoints. If any of the three points is a local minimum point and has least function value, the function terminates with a solution; otherwise, the point with least function value is used as the starting point.

From the starting point, for example x_c, the function value $f_c = f(x_c)$, the derivative value $g_c = g(x_c)$, and a new point x_n, defined by
$x_n = x_c - g_c$, are computed. The function $f_n = f(x_n)$ and the derivative $g_n = g(x_n)$ are then evaluated. If either $f_n \geq f_c$ or g_n has the opposite sign of g_c, then a minimum point exists between x_c and x_n, and an initial interval is obtained; otherwise, since x_c is kept as the point that has lowest function value, an interchange between x_n and x_c is performed. The secant method is then used to get a new point:

$$x_s = x_c - g_c\left(\frac{g_n - g_c}{x_n - x_c}\right)$$

Let $x_n \leftarrow x_s$. Repeat this process until an interval containing a minimum is found or one of the following convergence criteria is satisfied:

Criterion 1: $| x_c - x_n | \leq \varepsilon_c$

Criterion 2: $| g_c | \leq \varepsilon_g$

where $\varepsilon_c = \max \{1.0, | x_c |\} * \varepsilon$, ε is a relative error tolerance and ε_g is a gradient tolerance.

When convergence is not achieved, a cubic interpolation is performed to obtain a new point. The function and derivative are then evaluated at that point; accordingly, a smaller interval that contains a minimum point is chosen. A safeguarded method is used to ensure that the interval be reduced by at least a fraction of the previous interval. Another cubic interpolation is then performed, and this function is repeated until one of the stopping criteria is met.

Example

In this example, the minimum point of $f(x) = e^x - 5x$ is found.

```
.RUN
    ; Define the function to be used.
FUNCTION f, x
RETURN, EXP(x) - 5 * x
END
    % Compiled module: F.
xmin = FMIN('f', -100, 100)
    ; Call FMIN to compute the minimum.
PM, xmin
    1.60943
```

```
x = 10 * FINDGEN(100)/99 - 5

!P.Font = 0

PLOT, x, f(x), Title = '!8f(x) = e!Ex!N-5x!3', $
   XTitle = 'x', YTitle = 'f(x)'
```

; Plot results (*Figure 2-12*).

```
OPLOT, [xmin], [f(xmin)], Psym = 6

str = '(' + STRCOMPRESS(xmin) + ',' + $
   STRCOMPRESS(f(xmin)) + ')'

OPLOT, [xmin], [f(xmin)], Psym = 6

XYOUTS, -5, 80, 'Minimum point:!C' + str, $
   Charsize = 1.2
```

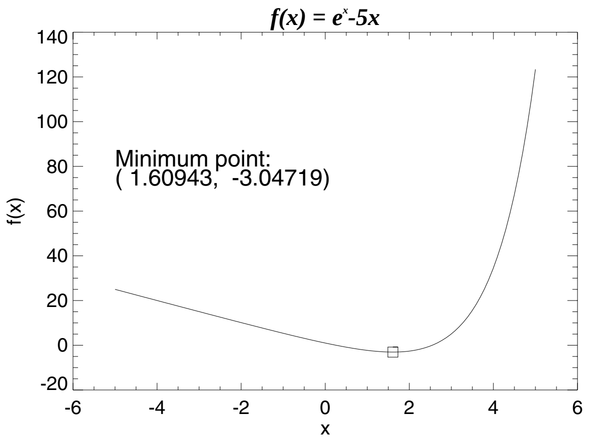


Figure 2-12 Plot showing the minimum point on a curve.

See Also

FMINV

FMINV Function

Minimizes a function $f(x)$ of n variables using a quasi-Newton method.

Usage

result = FMINV(*f*, *n*)

Input Parameters

f — A scalar string specifying a user-supplied function to evaluate the function to be minimized. The *f* function accepts the following parameter and returns the computed function value at the point:

> ***x*** — The point at which the function is evaluated.

n — The number of variables.

Returned Value

result — The minimum point *x* of the function. If no value can be computed, NaN is returned.

Keywords

Double — If present and nonzero, double precision is used.

FScale — A scalar containing the function scaling. Keyword *FScale* is used mainly in scaling the gradient. See keyword *Tol_Grad* for more detail. (Default: *FScale* = 1.0)

FValue — The name of a variable into which the value of the function at the computed solution is stored.

Grad — A scalar string specifying a user-supplied function to compute the gradient. This function accepts the following parameter and returns the computed gradient at the point:

> ***x*** — The point at which the gradient is evaluated.

Ihess — The Hessian initialization parameter. If *Ihess* is zero, the Hessian is initialized to the identity matrix; otherwise, it is initialized to a diagonal matrix containing $\max(f(t), f_s) * s_i$ on the diagonal, where t = *XGuess*, f_s = *FScale*, and s = *XScale*. (Default: *Ihess* = 0)

Itmax — The maximum number of iterations. (Default: *Itmax* = 100)

Max_Evals — The maximum number of function evaluations. (Default: *Max_Evals* = 400)

Max_Grad — The maximum number of gradient evaluations. (Default: *Max_Grad* = 400)

Max_Step — The maximum allowable step size.
(Default: *Max_Step* = 1000max (ε_1, ε_2), where

$$\varepsilon_1 = \sqrt{\sum_{i=1}^{n} (s_i t_i)^2}$$

$$\varepsilon_2 = \|s\|_2,$$

s = *XScale*, and t = *XGuess*)

N_Digit — The number of good digits in the function. (Default: machine dependent)

Tol_Grad — The scaled gradient tolerance. The *i*-th component of the scaled gradient at *x* is calculated as

$$\frac{|g_i| \times \max(|x_i|, 1/s_i)}{\max(|f(x)|, f_s)} \text{ ,where } g = \nabla f(x),$$

s = *XScale*, and f_s = *FScale*.

(Default: *Tol_Grad* = $\varepsilon^{1/2}$ ($\varepsilon^{1/3}$ in double), where ε is the machine precision)

Tol_Rfcn — The relative function tolerance.
(Default: *Tol_Rfcn* = max (10^{-10}, $\varepsilon^{2/3}$), max (10^{-20}, $\varepsilon^{2/3}$) in double)

Tol_Step — The scaled step tolerance. The *i*-th component of the scaled step between two points *x* and *y* is computed as

$$\frac{|x_i - y_i|}{\max(|x_i|, 1/s_i)}$$

where s = *XScale*. (Default: *Tol_Step* = $\varepsilon^{2/3}$)

XGuess — An array with *n* components containing an initial guess of the computed solution. (Default: *XGuess* (*) = 0)

XScale — An array with *n* components containing the scaling vector for the variables. Keyword *XScale* is used mainly in scaling the gradient and the distance between two points. See keywords *Tol_Grad* and *Tol_Step* for more detail. (Default: *XScale* (*) = 1.0)

Discussion

FMINV uses a quasi-Newton method to find the minimum of a function $f(x)$ of n variables. The problem is stated below.

$$\min_{x \in \mathbf{R}^n} f(x)$$

Given a starting point x_c, the search direction is computed according to the formula

$$d = -B^{-1} g_c$$

where B is a positive definite approximation of the Hessian and g_c is the gradient evaluated at x_c.

A line search is then used to find a new point

$$x_n = x_c + \lambda d \; , \; \lambda > 0$$

such that

$$f(x_n) \le f(x_c) \alpha g^T d$$

where $\alpha \in (0, 0.5)$. Finally, the optimality condition $\|g(x)\| \le \varepsilon$ is checked, where ε is a gradient tolerance.

When optimality is not achieved, B is updated according to the BFGS formula

$$B \leftarrow B - \frac{Bss^T B}{s^T Bs} + \frac{yy^T}{y^T s}$$

where $s = x_n - x_c$ and $y = g_n - g_c$. Another search direction is then computed to begin the next iteration. For more details, refer to Dennis and Schnabel (1983).

In this implementation, the first stopping criterion for FMINV occurs when the norm of the gradient is less than the given gradient tolerance *Tol_Grad*. The second stopping criterion for FMINV occurs when the scaled distance between the last two steps is less than the step tolerance *Tol_Step*.

Since by default, a finite-difference method is used to estimate the gradient for some single-precision calculations, an inaccurate estimate of the gradient may cause the algorithm to terminate at a noncritical point. In such cases, high-precision arithmetic is recommended or keyword *Grad* is used to provide more accurate gradient evaluation.

Example

In this example, the function
$f(x) = 100 (x_2 - x_1^2) ^2 + (1 - x_1)^2$ is minimized using FMINV.

```
.RUN
    ; Define the function.
FUNCTION f, x
xn = x
xn(0) = x(1) - x(0)^2
xn(1) = 1 - x(0)
RETURN, 100 * xn(0)^2 + xn(1)^2
END
% Compiled module: F.
xmin = FMINV("f", 2)
    ; Compute the minimum.
PM, xmin, Title = 'Solution:'
    Solution:
    0.999986
    0.999971
PM, f(xmin), Title = 'Function value:'
    Function value:
    2.09543e-10
```

See Also

FMIN

FREQRESP_S Function

Evaluates the frequency response of an analog filter for a given set of points on the complex plane using Horner's method.

Usage

result = FREQRESP_S(*h*, *pts*)

Input Parameters

h — An analog filter structure. This analog filter is expected to be a rational polynomial in positive powers of *s*.

pts — One or more points. (Most likely a complex array; however, this parameter may also be a scalar, a float, or an integer array.)

Returned Value

result — The value of filter $h = H(s)$ at a given set of points *s* on the complex plane ($s_i = (x_i + jy_i)$). The dimension of *result* is the same as the input parameter *pts*.

Keywords

None.

Discussion

FREQRESP_S uses Horner's method to evaluate the numerator and denominator polynomials at each point in the input array, and then divides the results. Floating underflow and overflow cause the usual error messages.

The most common use of FREQRESP_S is to determine the frequency response of an analog transfer function. This is accomplished by evaluating $s = j\omega$ for values of the real variable ω.

Example

This example computes the frequency response of a first order analog lowpass filter, $H(s) = 0.5/(s + 0.5)$.

```
b = 0.5
```

```
a = [0.5, 1]
h = FILTSTR(b, a)
omega = COMPLEX(FLTARR(100), 5.*INDGEN(100)/99.)
PLOT, ABS(omega), ABS(FREQRESP_S(h, omega)
```

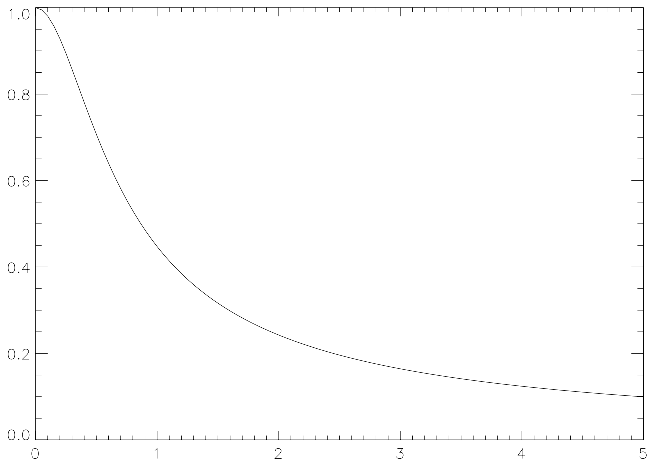


Figure 2-13 Plot of the frequency response of first order analog lowpass filter.

See Also

FILTSTR, FREQRESP_Z

FREQRESP_Z Function

Evaluates the frequency response of a digital filter on the unit circle.

Usage

result = FREQRESP_Z(*h* [, *npoints*])

Input Parameters

h — A digital filter structure.

npoints — (optional) The number of equally spaced points on the unit circle. (Default: *npoints* = 512)

Returned Value

result — The value of the transfer function *H*(*z*) evaluated at a set of frequency points

$${}_k = e^{j\pi f_k}.$$

Keywords

Infreq — One or more real values indicating the normalized frequencies at which the function is evaluated.

Outfreq — If set, returns an array of normalized frequencies at which the transfer function was evaluated.

Slow — If present and nonzero, FREQRESP_Z evaluates the numerator and the denominator at each point instead of performing the FFT.

NOTE If *Slow* is set, FREQRESP_Z will not perform the FFT, even when the FFT would normally be used.

Discussion

FREQRESP_Z evaluates the digital transfer function

$$H(z) = B(z)/A(z)$$

at a set of points

$${}_k = e^{j\pi f_k}.$$

If an empty variable is used with the *Outfreq* keyword, as in the command

```
response = FREQRESP_Z(h, Outfreq = freq)
```

the values of the normalized frequencies will be passed back into the variable specified by the keyword *Outfreq*.

If *Infreq* is set to a scalar or array, those normalized frequencies f_k are used to generate z_k values. If *Infreq* isn't set but *npoints* is, FREQRESP_Z generates that number of evenly spaced points (*z* values) around the top half of the unit circle. If neither keyword is used, the default value of 512 evenly spaced points is generated (around the top half of the unit circle).

If *Infreq* or *Slow* is used, the generated values and the filter structure *H*, are passed to FREQRESP_S and the results are returned.

Example

In this example, FREQRESP_Z is used to evaluate the frequency response of a lowpass filter (*Figure 2-14*).

```
h = IIRDESIGN(10, .5, /Butter)

response = FREQRESP_Z(h, Outfreq = f)

PLOT, f, ABS(response)
```

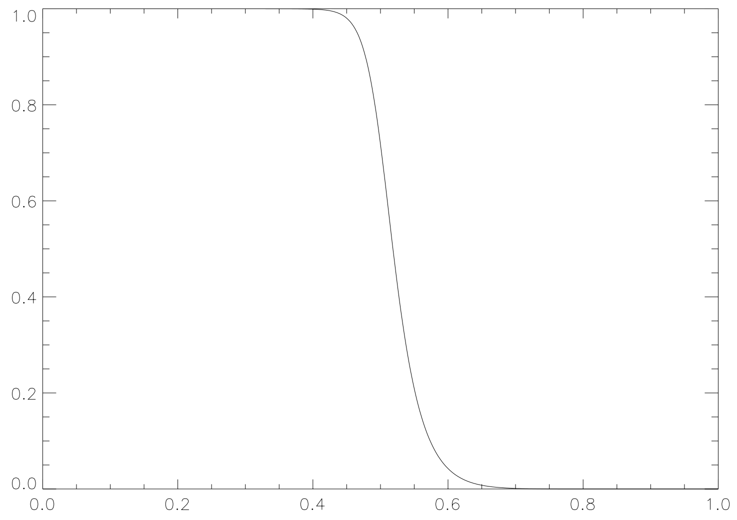


Figure 2-14 Plot of the frequency response of a lowpass filter.

See Also

FREQRESP_S

FREQTRANS Function

Performs frequency transformation of a lowpass prototype filter (in the normal filter structure) *H*(*z*) into another filter *G*(*z*) by replacing *z* with a stable all-pass filter *F*(*z*).

Usage

result = FREQTRANS(*h*[, *p*])

Input Parameters

h — A digital filter structure. If you want to use one of the four predefined standard transformations, this filter is assumed to be lowpass, by convention.

p — (optional) An array of polynomial coefficients in z^{-1}. This polynomial is optional, and will be generated for you if you use one of the standard transformations: lowpass, highpass, bandpass, or bandstop.

Returned Value

result — A filter structure containing the transformed filter.

Keywords

Bandpass — [*oldcutoff*, *lowcutoff*, *upcutoff*] Where *oldcutoff* is the cutoff frequency of the prototype lowpass filter; *lowcutoff* is the lower cutoff frequency; and *upcutoff* is the upper cutoff frequency.

Bandstop — [*oldcutoff*, *lowcutoff*, *upcutoff*] Where *oldcutoff* is the cutoff frequency of the prototype lowpass filter; *lowcutoff* is the lower cutoff frequency; and *upcutoff* is the upper cutoff frequency.

Highpass — [*oldcutoff*, *newcutoff*] Where *oldcutoff* is the cutoff frequency of the lowpass filter to be transformed; *newcutoff* is the cutoff frequency of the transformed filter returned.

Lowpass — [*oldcutoff*, *newcutoff*] Where *oldcutoff* is the cutoff frequency of the lowpass filter to be transformed; *newcutoff* is the cutoff frequency of the transformed lowpass filter returned.

Newname — If the *Newname* parameter isn't passed in, the new filter structure has the same name as the old one.

Sigma — Set to 1 to return a lowpass filter and –1 for a highpass filter. (Default: *Sigma* = 1)

Discussion

Keywords are used to specify one of four standard transformation types. Each of the keyword parameters used to call a standard transform accepts an array, and each array requires that a new cutoff frequency be specified. These new frequencies should be normalized between 0 and 1.

Once this is done, a complementary polynomial to $p(z)$ is created. The two polynomials are related as

$$\tilde{p}(z) = z^{-M} p(z^{-1}).$$

The transform itself is given by

$$G(z) = H(\sigma F(z)) = \frac{b(z)}{a(z)},$$

where

$$F(z) = \frac{p(z)}{\tilde{p}(z)},$$

is a stable all-pass filter and

$$p(z) = p_0 + p_1 z^{-1} + p_2 z^{-2} + \ldots.$$

The numerator polynomial p is a polynomial in z^{-1}, such as the FREQTRANSDESIGN function creates.

CAUTION Do not use a transformation order that is very high. The resulting high order filter may be numerically unstable.

NOTE If polynomial p is obtained using FREQTRANSDESIGN, the lowpass prototype filter $H(z)$ must have a normalized frequency cutoff of 0.5.

NOTE The polynomial p must be stable and pass the Schur-Cohn test. Otherwise, FREQTRANS will fail.

Example

This example generates a lowpass Butterworth filter and then moves the cutoff frequency (*Figure 2-15*).

```
fp = 0.2049
n = 8
h = IIRDESIGN(n, fp, /Butter)
    ; Define an eighth-order Butterworth filter.
newfp = 0.4
result = FREQTRANS(h, Lowpass = [fp, newfp])
    ; Now, move the cutoff frequency from 0.2049 to 0.4.
```

```
resp = ABS(FREQRESP_Z(result, Outfreq = freq))
newresp = ABS(FREQRESP_Z(h, Outfreq = freq))
PLOT, freq, resp
```
; Plot the original lowpass Butterworth filter frequency response (*Figure 2-15*).

```
OPLOT, freq, newresp
```
; Plot the frequency response with the shifted cutoff frequency (*Figure 2-15*).

```
REFLINES, [fp, newfp]
```

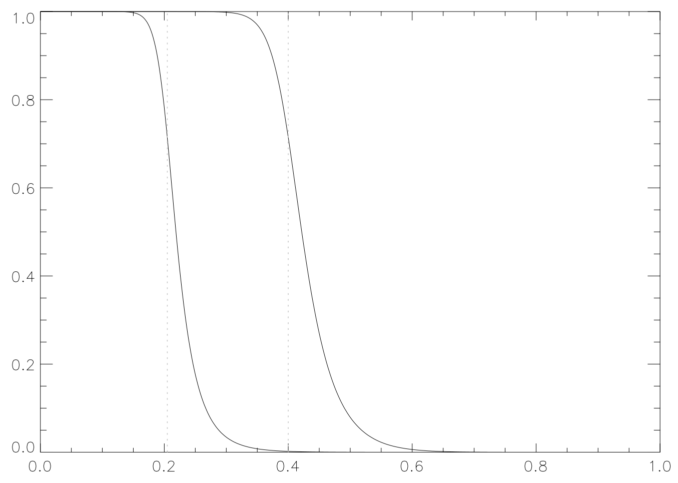


Figure 2-15 A lowpass Butterworth filter with a shifted cutoff frequency.

See Also

FREQTRANSDESIGN, IIRDESIGN

For Additional Information

Parks and Burrus, 1987.

Proakis and Manolakis, 1992.

Roberts and Mullis, 1987.

FREQTRANSDESIGN Function

Creates the numerator polynomial of a stable all-pass filter. The resulting filter is used to perform a frequency transformation of a lowpass prototype filter into a multiple bandpass filter.

Usage

result = FREQTRANSDESIGN(*phi*)

Input Parameters

phi— An array of normalized cutoff frequencies (between 0 and 1).

Returned Value

result — An array of polynomial coefficients.

Keywords

Lowstop — If present and nonzero, the transformation creates a lowstop filter. The filter passed into FREQTRANS must be a highpass filter.

Discussion

NOTE The transformation generated by FREQTRANSDESIGN must be applied to a lowpass filter with a cutoff frequency of 0.5 (normalized) or the transformation won't work correctly.

CAUTION Using this function to generate polynomials of order 13 or greater is not advised.

The *result* polynomial is passed to FREQTRANS along with a lowpass digital filter to create a multiple bandpass filter.

The algorithm used in FREQTRANSDESIGN comes from Franchitti, 1985.

Example

This example creates a lowpass filter and then transforms that into a multiple band-pass filter.

```
h = IIRDESIGN(15, 0.5, /Butter)
    ; Defines an order-15 Butterworth filter with normalized cutoff
    ; frequency of 0.5.

phi = [0.1, 0.4, 0.6, 0.7, 0.8]
    ; Defines the new cutoff frequencies.

!X.Ticks = 7

!X.Style = 1

!P.Charsize = 1.5

!Y.Range = [0,1.2]

!X.Tickv = [phi(0), phi(1), 0.5, phi(2), phi(3), $
   phi(4), 1.0]
        ; Set up some plotting parameters.

p = FREQTRANSDESIGN(phi)
    ; Construct the frequency transformation polynomial.

result = ABS(FREQRESP_Z(FREQTRANS(h, p), Outfreq = f))
        ; Transform the filter and compute the frequency response in one step.

ticks = ['!4u!6!l1', '!4u!6!l2', '0.5', $
   '!4u!6!l3', '!4u!6!l4', '!4u!6!l5', '1.0']

!X.Tickname = ticks

PLOT, f, result, Thick = 2
    ; Plot the frequency response of the transformed filter (see Figure 2-16).

OPLOT, f, ABS(FREQRESP_Z(h)), Thick = 2, Linestyle = 1
        ; Plot the response of the original filter (Figure 2-16).
REFLINES, phi
    ; Plot some lines at the cutoff frequencies (Figure 2-16).
```

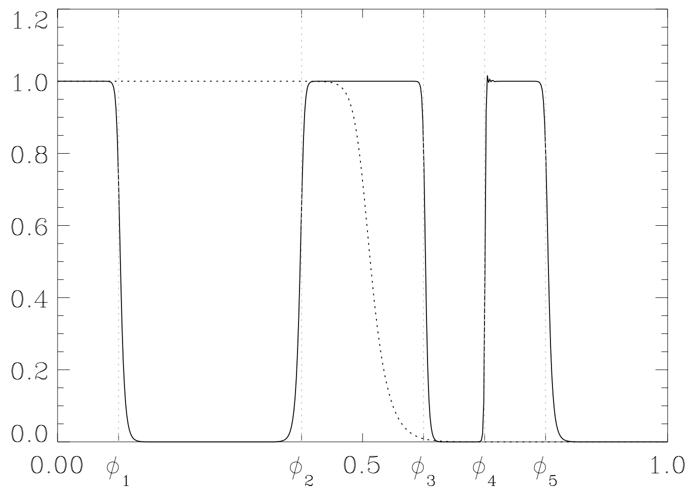


Figure 2-16 A plot of the original lowpass Butterworth filter, and the transformed multiple bandpass filter. Reference lines indicate the cutoff frequencies used in the FREQTRANS-DESIGN multiple bandpass example.

See Also

FREQTRANS

For Additional Information

Franchitti, 1985.

Roberts and Mullis, 1987, pp. 202 – 207.

HILBERT Function

Constructs a Hilbert transformation.

Usage

result = HILBERT(*x* [, *d*])

Input Parameters

x — The array to be transformed. Can be of either floating-point or complex data type, and can contain any number of elements.

d — (optional) A flag to indicate the direction of rotation:

+1	Shifts the array +90 degrees.
–1	Shifts the array –90 degrees.

Returned Value

result — A complex data type, the value of which is the Hilbert transform of *x*, having the same dimensions as *x*.

Keywords

None.

Discussion

A Hilbert transform is a series of numbers in which all periodic components have been phase-shifted by 90 degrees. Angle shifting is accomplished by multiplying or dividing by the complex number
j = (0.000, 1.000).

A Hilbert series has the interesting property that the correlation between it and its own Hilbert transform is mathematically zero.

NOTE The HILBERT function creates a Hilbert matrix by computing the fast Fourier transform of the data with the PV‑WAVE FFT function and shifting the first half of the transform products by +90 degrees and the second half by –90 degrees. The constant elements of the transform are not changed.

The shifted array is then submitted to the PV‑WAVE FFT function for the transformation back to the time-domain. Before it is returned, the output is divided by the number of elements in the array to correct for the multiplication effect characteristic of the FFT algorithm.

Example

```
!P.multi = [0, 1, 3]

a = FINDGEN(1000)

sine_wave = SIN(a/(MAX(a)/(2 * !Pi)))
        ; Create a sine wave.

PLOT, sine_wave
        ; Plot the sine wave (Figure 2-17 (a)).

OPLOT, HILBERT(sine_wave, -1)
        ; Plot the sine wave phase-shifted to the right by 90 degrees
        ; (Figure 2-17 (a)).

rand = RANDOMN(seed, 1000) * 0.05
        ; Create an array of random numbers to mimic a noisy signal.

PLOT, rand
        ; Plot the random numbers (Figure 2-17 (b)).

sandwich = [sine_wave, rand, sine_wave]
        ; Sandwich the random data between two sine waves.

PLOT, sandwich, XStyle=1
        ; Plot the two sine waves with the random noise in the middle,
        ; thereby turning them into a single signal (Figure 2-17 (c)).

OPLOT, HILBERT(sandwich, -1)
        ; Plot the sandwiched wave forms. Note that the sine waves are
        ; phase-shifted to the right by 90 degrees, while the noise data
        ; has not shifted at all, but rather has been distorted vertically
        ; (its amplitude) by the effect of the two adjacent phase-shifted sine
        ; waves. This is because the sine waves and the noise data were set
        ; up to be a single signal (Figure 2-17 (c)).
```

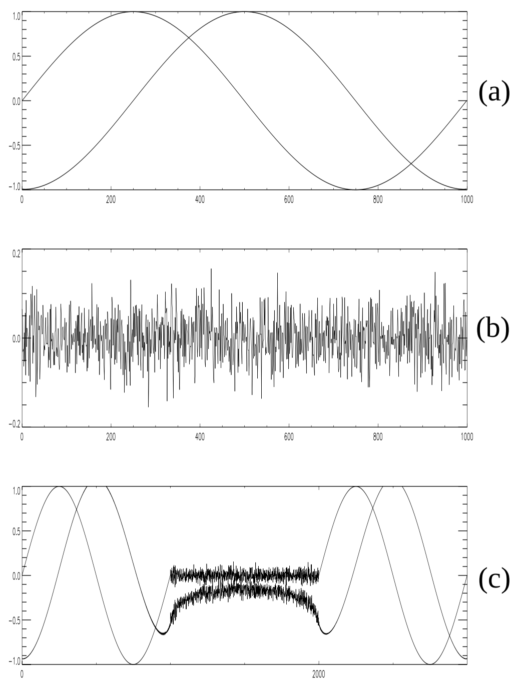


Figure 2-17 (a) Sine wave and 90-degree phase shifted sine wave. (b) Random noise plot. (c) Sandwiched sine waves with random noise and the HILBERT transform of that signal.

See Also

In the *PV*-*WAVE Reference*:

FFT

IIRDESIGN Function

Designs Butterworth, Chebyshev Type I, Chebyshev Type II, and elliptic lowpass infinite impulse response (IIR) digital filters.

Usage

result = IIRDESIGN(*n*, *fp*, /*Butter*)

result = IIRDESIGN(*n*, *fp*, *rp*, /*Cheby1*)

result = IIRDESIGN(*n*, *fs*, *rs*, /*Cheby2*)

result = IIRDESIGN(*n*, *fp*, *rp*, *rs*, /*Ellip*)

Input Parameters

n — The filter order.

fp — The pass-band frequency edge.

fs — The stop-band frequency edge.

rp — The pass-band ripple.

rs — The stop-band ripple.

Returned Value

result — A filter structure containing the coefficients of the IIR filter.

Keywords

Butter — If present and nonzero, designs a Butterworth filter.

Cheby1 — If present and nonzero, designs a Chebyshev Type I filter.

Cheby2 — If present and nonzero, designs a Chebyshev Type II filter.

Ellip — If present and nonzero, designs an elliptic filter.

Discussion

This function designs Butterworth, Chebyshev types I and II, and elliptic IIR digital lowpass filters. The different filter approximations are illustrated in *Figure 2-18*.

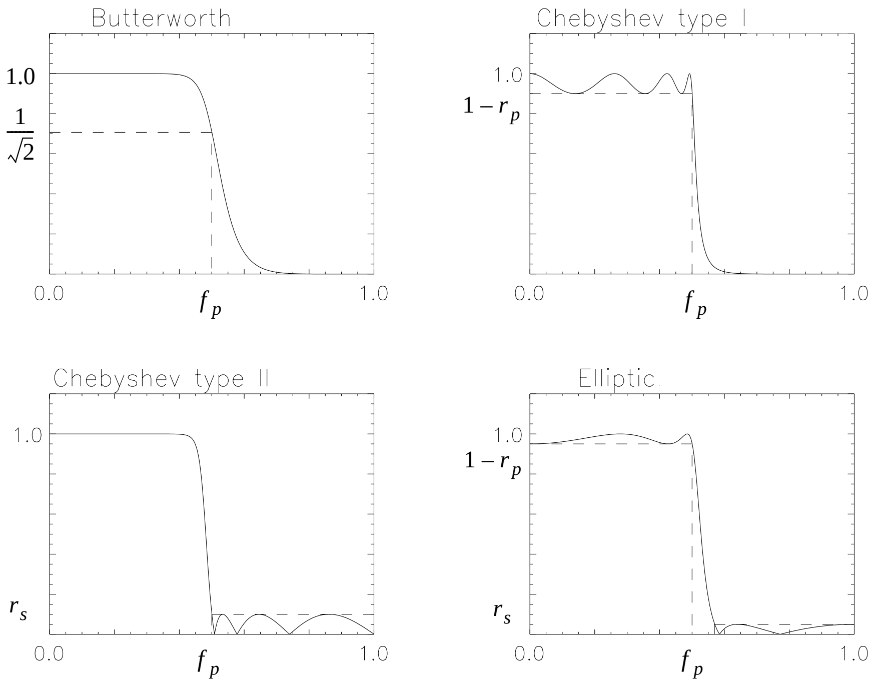


Figure 2-18 Filter approximations for Butterworth, Chebyshev types I and II, and elliptic IIR digital lowpass filters.

The Butterworth filter is maximally flat in the pass and stop bands. This filter is parameterized by its order (*n*) and pass-band frequency edge (f_p) defined by

$$\left|H(e^{j\pi f_p})\right| = \frac{1}{\sqrt{2}}.$$

The Chebyshev Type I filter has equal ripple in the pass band. This filter is parameterized by its order (*n*), pass-band ripple (r_p), and pass-band frequency edge (f_p) defined by

$$\left|H(e^{j\pi f_p})\right| = 1 - r_p.$$

The Chebyshev Type II filter has equal ripple in the stop band. This filter is parameterized by its order (n), the stop-band ripple (r_s), stop-band frequency edge (f_s) defined by

$$\left|H(e^{j\pi f_s})\right| = r_s .$$

The elliptic filter has equal ripple in both the pass band and the stop band. This filter is parameterized by its order (n), the pass-band ripple (r_p), and stop-band ripple (r_s), the pass-band frequency edge (f_p) defined by

$$H(e^{j\pi f_p})\Big| = 1 - r_p.$$

The four filter types are obtained by first designing an analog lowpass prototype filter using the techniques discussed in Parks and Burrus (1987) and then using the bilinear transform to obtain an digital lowpass prototype filter.

The lowpass filters obtained using IIRDESIGN can be transformed into a highpass, bandpass, or bandstop filter using the function FREQTRANS, if desired.

The minimum filter order required to meet a set of specifications for the filters designed by IIRDESIGN may be determined by IIRORDER.

Example 1

In this example, a Butterworth filter is designed and the resulting frequency response is plotted (*Figure 2-19*).

```
n = 7

fp = 0.5

h = IIRDESIGN(n, fp, /Butter)

hf = FREQRESP_Z(h, Outfreq = f)

PLOT, f, ABS(hf), YRange = [0, 1.2], $
   Title = 'Butterworth Magnitude', $
   XTitle = 'Frequency'

OPLOT, [0, fp], SQRT(0.5)*[1, 1], Linestyle = 2

OPLOT, [fp, fp], SQRT(0.5)*[1, 0], Linestyle = 2
```

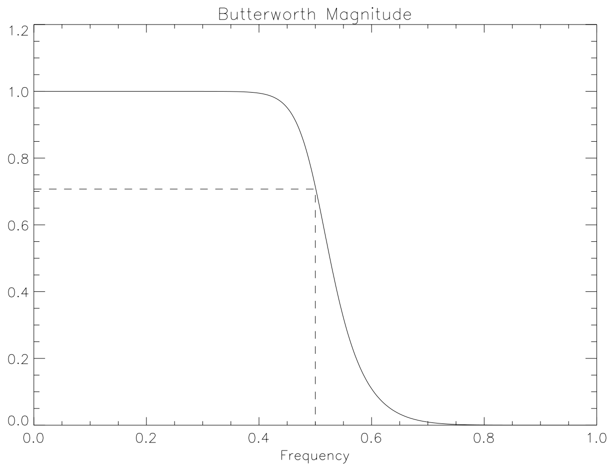


Figure 2-19 Frequency response from a seventh-order Butterworth filter.

Example 2

A Chebyshev Type I filter is designed and the resulting frequency response is plotted in *Figure 2-20*.

```
n = 7

fp = 0.5

rp = .2

h = IIRDESIGN(n, fp, rp, /Cheby1)

hf = FREQRESP_Z(h, Outfreq = f)

PLOT, f, ABS(hf), $
   Title = 'Chebyshev Type I Magnitude', $
   XTitle = 'Frequency'

OPLOT, [0, fp], (1 - rp)*[1, 1], Linestyle = 2

OPLOT, [fp, fp], (1 - rp)*[1, 0], Linestyle = 2
```

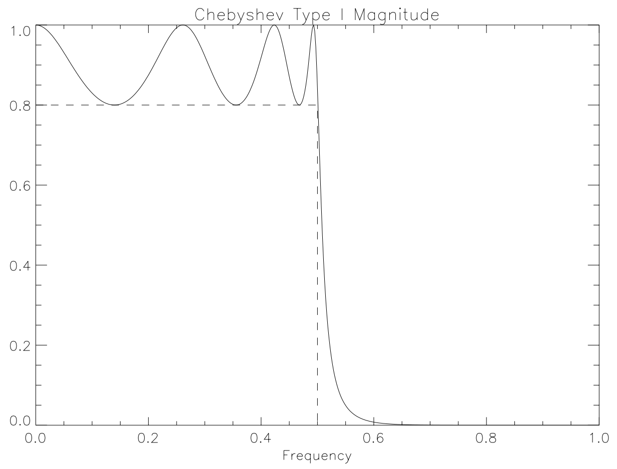


Figure 2-20 Frequency response from a seventh-order Chebyshev type I filter.

Example 3

A Chebyshev Type II filter, also known as an inverse Chebyshev filter is designed in this example. The frequency response is plotted in *Figure 2-21*.

```
n = 7

fs = 0.5

rs = 0.2

h = IIRDESIGN(n, fs, rs, /Cheby2)

hf = FREQRESP_Z(h, Outfreq = f)

PLOT, f, ABS(hf), YRange = [0, 1.2], $
   Title = 'Chebyshev Type II Magnitude', $
   XTitle = 'Frequency'

OPLOT, [fs, 1], rs*[1, 1], Linestyle = 2

OPLOT, [fs, fs], rs*[1, 0], Linestyle = 2
```

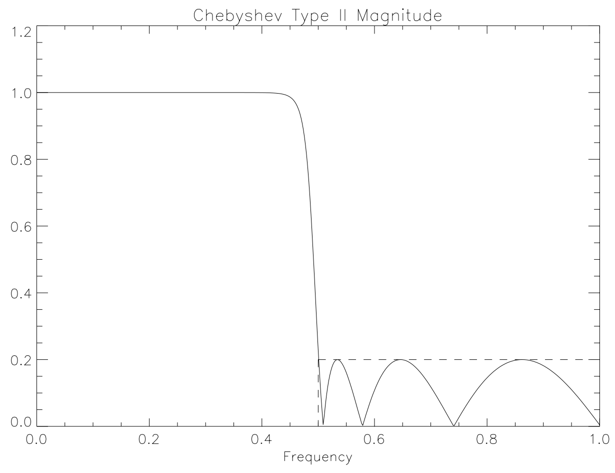


Figure 2-21 Frequency response from a seventh-order Chebyshev type II filter.

Example 4

In this example, an elliptic filter is designed and the frequency response is plotted (*Figure 2-22*).

```
n = 4

fp = .5

rp = .05

rs = .05

h = IIRDESIGN(n, fp, rp, rs, /Ellip)

hf = FREQRESP_Z(h, Outfreq = f)

PLOT, f, ABS(hf), Title = 'Elliptic Magnitude', XTitle = 'Frequency'

YRange = [0., 1.2], YStyle = 1

OPLOT, [0, fp], [1-rp, 1-rp], Linestyle = 2

OPLOT, [fp, fp], [0, 1-rp], Linestyle = 2

OPLOT, [fp, 1], [rs, rs], Linestyle = 2
```

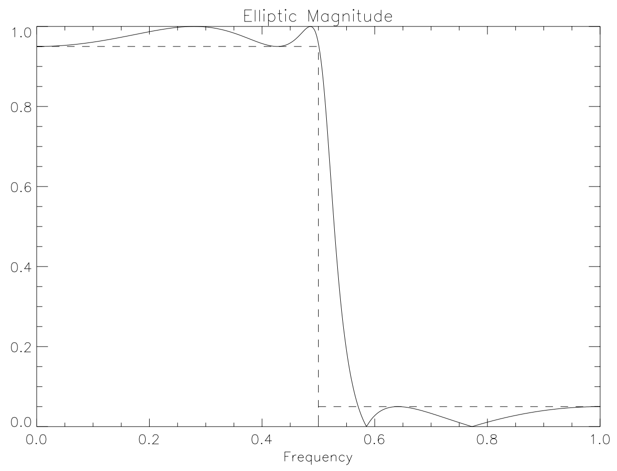


Figure 2-22 Frequency response from a fourth-order elliptic filter.

See Also

FREQTRANS, FREQTRANSDESIGN, IIRORDER

For Additional Information

Parks and Burrus, 1987.

IIRFILT Function

Applies an infinite impulse response (IIR) filter to a data sequence.

Usage

result = IIRFILT(*h*, *x*)

Input Parameters

h — A digital filter structure containing the filter coefficients.

x — A one-dimensional array containing the data to be filtered.

Returned Value

result — A one-dimensional array containing the filtered data. The returned array is the same dimension as the input array *x*.

Keywords

Forward_back — If present and nonzero, the data sequence is forward-backward filtered.

Discussion

IIRFILT realizes the digital filter

$$H(z) = \frac{B(z)}{A(z)} = \frac{b_0 + b_1 z^{-1} + \ldots + b_M z^{-M}}{1 + a_1 z^{-1} + \ldots + a_N z^{-N}}$$

using the transposed direct form II filter structure shown in *Figure 2-23*.

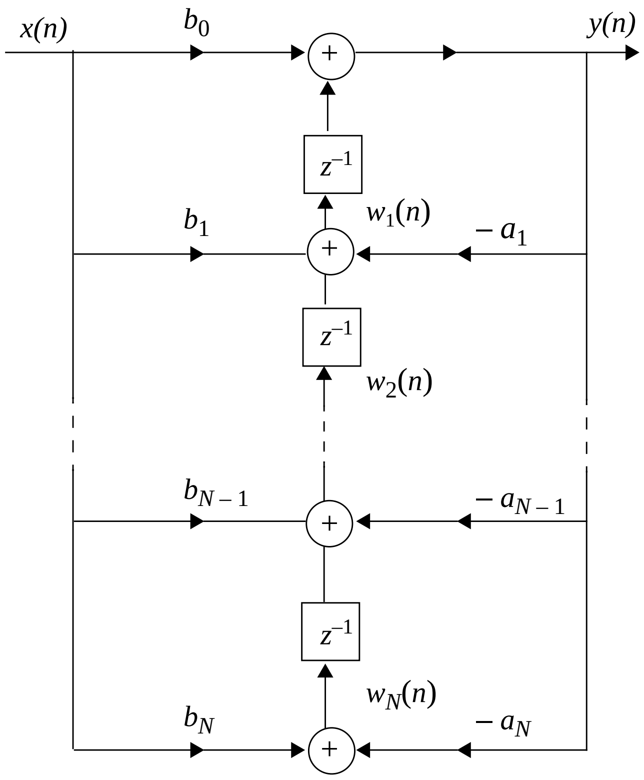


Figure 2-23 Signal flow graph for IIR filter realization.

The difference equations associated with this filter structure are given by

$$y(n) = w_1(n-1) + b_0x(n)$$

$$w_k(n) = w_{k+1}(n-1) - a_ky(n) + b_kx(n)\ , \qquad k = 1, 2, ..., \max\{M, N\}$$

$$w_N(n) = b_Nx(n) - a_Ny(n)$$

When the keyword *Forward_back* is specified, this function filters the data sequence in both the forward and backward directions. Forward-backward filtering is equivalent to applying the filter

$$H(z) = \frac{B(z)B(z^{-1})}{A(z)A(z^{-1})}.$$

The algorithm used to accomplish the forward-backward filter operation first solves the equation

$$\frac{B(z)B(z^{-1})}{A(z)A(z^{-1})} = \frac{C(z)}{A(z)} + \frac{C(z^{-1})}{A(z^{-1})}$$

for $C(z)$ using the Jury algorithm discussed in Demeure and Mullis (1990). The signal is then filtered according to the signal flow diagram shown in *Figure 2-24*.

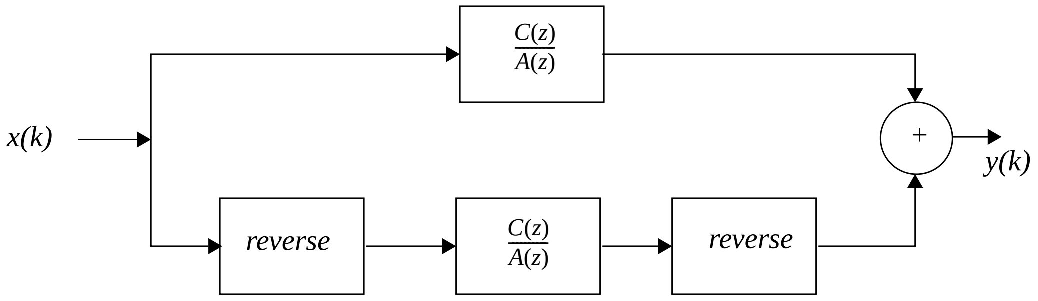


Figure 2-24 Signal flow diagram for forward-backward filtering.

The “reverse” operation shown in the figure takes a sequence

$$x(k), \qquad k = 0, 1, ..., L$$

and replaces it with the sequence

$$x(L-k), \qquad k = 0, 1, ..., L \;.$$

Forward-backward filtering is typically used to obtain zero-phase response from an IIR filter.

NOTE Each of these methods assumes that the constant term in the denominator polynomial of the transfer function is unity. To satisfy that assumption, both the numerator and denominator polynomials are scaled by a_0.

Example

In this example, IIRFILT is used to perform causal and anti-causal filtering.

```
!P.multi = [0, 1, 2]
x = ((INDGEN(1024)+64) MOD 256) GT 128
```

; Generate a square wave.

```
h = IIRDESIGN(5, 0.25, 0.01, 0.01, /Ellip)
```

; Design an elliptic lowpass filter.

```
PLOT, IIRFILT(h, x), Title = 'Causal Filtering', XTitle = 'Time'
```

; Causal filtering of the square wave (*Figure 2-25* (a)).

```
PLOT, IIRFILT(h, x, /Forward_back), $
   Title = 'Forward-Backward Filtering', $
   XTitle = 'Time'
```

; Anti-causal (forward-backward) filtering of the square wave; (*Figure 2-25* (b)).

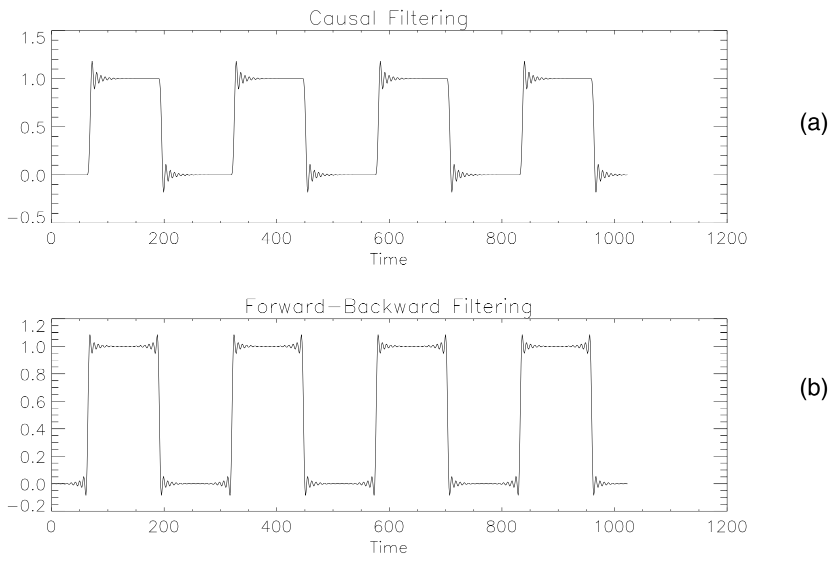


Figure 2-25 (a) Causal and (b) anti-causal filtering of a square wave accomplished using the IIRFILT function.

See Also

FILTER

For Additional Information

Proakis and Manolakis, 1992.

Demeure and Mullis, 1990.

IIRLS Function

Approximates time-domain or frequency-domain least squares infinite impulse response (IIR) digital filters.

Usage

result = IIRLS(*m*, *n*, *h*[, *l*])

Input Parameters

m — The order of the filter numerator polynomial.

n — The order of the filter denominator polynomial.

h — An array containing samples of the impulse response or frequency response.

l — (optional) The total number of frequency response samples used in the design. This parameter is required if the *Freqsample* keyword is specified.

Returned Value

result — A filter structure containing the filter approximation.

Keywords

Freqsample — If present and nonzero, a frequency-domain least squares method is used.

NOTE The optional input parameter *l* must be used when *Freqsample* is specified.

Prony — If present and nonzero, Prony's time-domain least squares method is used.

Discussion

IIRLS determines the numerator and denominator coefficients of an IIR filter

$$H(z) = \frac{B(z)}{A(z)} = \frac{b_0 + b_1 z^{-1} + \ldots + b_M z^{-M}}{1 + a_1 z^{-1} + \ldots + a_N z^{-N}} = \sum_{n=0}^{\infty} h(n) z^{-n}$$

to approximate a given set of impulse response samples

$$h(k), \quad k = 0, 1, ..., K;$$

or uniform set of frequency response samples

$$H_k = H(e^{j\pi f_k}),$$

where the frequency samples are selected as

$$f_k = \frac{k}{L+1}, \quad \text{k = 0, 1, ..., L.}$$

If *Prony* is specified, Prony's method for time-domain design of FIR filters is used. This method is based on noting that the rational transfer function may be rewritten

$$B(z) = H(z)A(z) .$$

Using the first $K + 1$ terms of the impulse response $h(n)$, the inverse z-transform of this equation can be written

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ \vdots \\ b_M \\ 0 \\ \vdots \\ 0 \end{bmatrix} = \begin{bmatrix} h(0) & 0 & 0 & \cdots & 0 \\ h(1) & h(0) & 0 & & \\ h(2) & h(1) & h(0) & & \\ \vdots & & & & \\ h(M) & & & & \\ \vdots & & & & \\ h(K) & & \cdots & & h(K-N) \end{bmatrix} \begin{bmatrix} 1 \\ a_1 \\ a_2 \\ \vdots \\ a_N \end{bmatrix}$$

Given $K \geq M + N$ impulse response samples, Prony's method selects the filter coefficients to minimize the least-squares equation error of the above matrix equation. If $K = M + N$ and the impulse response samples are consistent with a rational transfer function, this method will produce a filter with an impulse response that matches the K-given impulse response samples.

If *Freqsample* is specified, the frequency-domain least squares method is used. This method is based on noting that if we have $L + 1$ uniform samples of the frequency response, the sampled rational transfer function may be rewritten

$$B_k = H_k A_k .$$

In this relationship, H_k is the frequency response sample defined previously,

$$B_k = DFT\{b_n\}$$

and

$$A_k = DFT\{a_n\} \ ,$$

and both discrete Fourier transforms (DFT) are taken over L points. The inverse DFT of the above equation can be written

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ \vdots \\ b_M \\ 0 \\ \vdots \\ 0 \end{bmatrix} = \begin{bmatrix} h_0 & h_L & h_{L-1} & \cdots & h_2 & h_1 \\ h_1 & h_0 & h_L & & & h_2 \\ h_2 & h_1 & h_0 & & & \\ \vdots & & & \ddots & & \\ & & & & & \\ h_L & & & & & h_0 \end{bmatrix} \begin{bmatrix} 1 \\ a_1 \\ a_2 \\ \vdots \\ a_N \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

where matrix elements h_n are defined as

$$H_k \xleftrightarrow{\textit{inverse DFT}} h_n$$

Given $L \geq M + N$ uniform frequency response samples, IIRLS selects the filter coefficients to minimize the least squares equation error of the matrix equation. If $L = M + N$ and the frequency response samples are consistent with a rational transfer function, this method will produce a filter with frequency response that interpolates the L-given frequency response samples.

IIRLS only produces filters with real coefficients. This requires that the sequence h_n be real. The values of H_k are therefore forced to satisfy the standard symmetry properties of the DFT to ensure that h_n is real. If the total number of frequency samples L used in technique is odd, the number of frequency response samples H_k provided by the user must be greater than $(L + 1)/2$. If L is even, the number of samples must be greater than or equal to $(L + 2)/2$. Because of the forced symmetry, only the first $(L + 1)/2$ (for L odd) or $(L + 2)/2$ (for L even) elements of H_k are used in the routine. Any additional samples of H_k are ignored.

NOTE If the impulse or frequency response samples are consistent with an unstable filter, then an unstable filter will be returned by IIRLS. It is recommended that you verify the stability of the returned filter by applying SCHURCOHN to the denominator coefficients. It is also possible to stabilize an unstable filter without affecting its magnitude response by using the P_STAB function.

Example 1

This example illustrates how IIRLS with the keyword *Prony* can be used to design a filter that matches a given set of $L + 1$ impulse response samples.

```
l = 6

h = IIRDESIGN(l/2, 0.5, /Butter)

desired_impul = IMPRESP(h, l+1)
   ; Design a filter and compute a set of impulse response samples.

PARSEFILT, h, name, numer, denom

PM, numer, Title = $
   'Original Filter Numerator Coefficients'
   Original Filter Numerator Coefficients
      0.16666667
      0.50000000
      0.50000000
      0.16666667

PM, denom, Title = $
   'Original Filter Denominator Coefficients'
   Original Filter Denominator Coefficients
      1.0000000
      0.0000000
      0.3333333
      0.0000000

hls = IIRLS(l/2, l/2, desired_impul, /Prony)
   ; Compute filter coefficients that match the desired impulse
   ; response samples.

PARSEFILT, hls, name, numerls, denomls

PM, numerls, Title = $
   'Least Squares Filter Numerator Coefficients'
   Least Squares Filter Numerator Coefficients
      0.16666667
      0.50000000
      0.50000000
      0.16666667
PM, denomls, $
   Title = 'Least Squares Filter ' + $
   'Denominator Coefficients'
   Least Squares Filter Denominator Coefficients
      1.0000000
      0.0000000
      0.33333333
      2.0744776e-17
```

Example 2

This example illustrates how IIRLS with the keyword *Freqsample* can be used to design a filter that interpolates a given set of *L* frequency response samples.

```
l = 8
h = IIRDESIGN(l/2, 0.5, /Butter)
f= -2*DINDGEN(l)/DOUBLE(l+1)
desired_freqsamples = FREQRESP_Z(h, Infreq = f)
   ; Design a filter and compute a set of frequency response samples.
PARSEFILT, h, name, numer, denom
PM, numer, Title = $
   'Original Filter Numerator Coefficients'
   Original Filter Numerator Coefficients
      0.093980851
      0.37592341
      0.56388511
      0.37592341
      0.093980851
PM, denom, Title = $
   'Original Filter Denominator Coefficients'
   Original Filter Denominator Coefficients
      1.0000000
      0.0000000
      0.48602882
      0.0000000
      0.017664801
hls = IIRLS(l/2, l/2, desired_freqsamples, l,$
   /Freqsample)
      ; Compute filter coefficients that match the desired frequency
      ; response samples.
```

```
PARSEFILT, hls, name, numerls, denomls
PM, numerls, Title = $
   'Least Squares Filter Numerator Coefficients'
   Least Squares Filter Numerator Coefficients
      0.093980844
      0.37592330
      0.56388476
      0.37592300
      0.093980683
PM, denomls, Title = $
   'Least Squares Filter Denominator Coefficients'
   Least Squares Filter Denominator Coefficients
       1.0000000
      -1.0287107e-06
       0.48602936
      -6.7503742e-07
       0.017664935
```

See Also

FIRLS, P_STAB, SCHURCOHN

For Additional Information

Parks and Burrus, 1987, sections 7.4 and 7.5.

IIRORDER Function

Determines the minimum filter order required for a Butterworth, Chebyshev Type I, Chebyshev Type II, or elliptic infinite impulse response (IIR) digital lowpass filter to satisfy a given set of pass-band frequency edge, stop-band frequency edge, pass-band ripple, and stop-band ripple constraints.

Usage

result = IIRORDER(*fp*, *fs*, *rp*, *rs*)

Input Parameters

fp — The pass-band frequency edge

fs — The stop-band frequency edge

rp — The pass-band ripple

rs — The stop-band ripple

NOTE In addition to these parameters, exactly one keyword must be set to select the filter type for which the filter order will be computed.

Returned Value

***result* —** A scalar long containing the computed filter order.

Keywords

Butter — If present and nonzero, computes the filter order for a Butterworth filter.

Cheby1 — If present and nonzero, computes the filter order for a Chebyshev Type I filter.

Cheby2 — If present and nonzero, computes the filter order for a Chebyshev Type II filter.

Ellip — If present and nonzero, computes the filter order for an elliptic filter.

Discussion

The pass-band frequency edge (f_p), stop-band frequency edge (f_s), pass-band ripple (r_p), and stop-band ripple (r_s) are defined as shown in *Figure 2-18* on page 99 and are discussed in the *Discussion* section of IIRDESIGN.

The equations which relate the frequency cutoff values, pass-band ripple, stop-band ripple, and filter order (n) are discussed in Rabiner and Gold, 1975, p. 241.

Example

This example illustrates how IIRORDER may be used to find the minimum required lowpass filter order.

```
fp = 0.5
fs = 0.55
rp = 0.1
rs = 0.1
```

; Select frequency band edges and ripple parameters.

```
PM, IIRORDER(fp, fs, rp, rs, /Butter), $
```

```
   Title = 'Butterworth Order'
   Butterworth Order
      20
PM, IIRORDER(fp, fs, rp, rs, /Cheby1), $
   Title = 'Chebyshev Type I Order'
   Chebyshev Type I Order
      7
PM, IIRORDER(fp, fs, rp, rs, /Cheby2), $
   Title = 'Chebyshev Type II Order'
   Chebyshev Type II Order
      7
PM, IIRORDER(fp, fs, rp, rs, /Ellip), $
   Title = 'Elliptic Order'
   Elliptic Order
      4
```

See Also

IIRDESIGN

For Additional Information

Parks and Burrus 1987, Chapter 7.

Rabiner and Gold 1975, p. 241.

IMPRESP Function

Computes the impulse response of a digital filter.

Usage

result = IMPRESP(*h*, *n*)

Input Parameters

h — A valid filter structure.

n — The total number of samples to generate.

Returned Value

result — A one-dimensional array containing the first *n* values of the impulse response.

Keywords

None.

Discussion

IMPRESP uses the IIRFILT function with the input sequence $x(n)$, defined by

$$x(n) = \begin{cases} 1, & n = 0 \\ 0, & \text{otherwise} \end{cases},$$

to compute the impulse response of a digital filter.

Example

In this example, the impulse response of a filter structure

$$H(z) = \frac{1}{(1 + az^{-1})},$$

is computed using IMPRESP, and the results along with the analytic solution are printed out.

```
a = -0.25

h = FILTSTR([1], [1, a])

ir = IMPRESP(h, 10)
```

; Results should be a^k for k = 0, 1, 2, ... when $H(z) = 1/(1 + az^{-1})$.

```
PM, [[ir], [.25^FINDGEN(10)]], $
   Title = 'Computed        Actual'
       Computed         Actual
      1.0000000       1.0000000
      0.25000000      0.25000000
      0.062500000     0.062500000
      0.015625000     0.015625000
      0.0039062500    0.0039062500
```

```
0.00097656250   0.00097656250
0.00024414062   0.00024414062
6.1035156e-05   6.1035156e-05
1.5258789e-05   1.5258789e-05
3.8146973e-06   3.8146973e-06
```

See Also

IIRFILT

INTFCN Function

Integrates a user-supplied function.

Usage

result = INTFCN(*f*, *a*, *b*)

Input Parameters

f — A scalar string specifying the name of the function to be integrated. The function *f* accepts one scalar parameter and returns a single scalar of the same type.

a — A scalar expression specifying the lower limit of integration.

b — A scalar expression specifying the upper limit of integration.

Returned Value

result — An estimate of the desired integral. If no value can be computed, NaN (not a number) is returned.

Global Keywords

The following seven keywords can be used in any combination with each method of integration except the non-adaptive method, which is triggered by the keyword *Smooth*. Because of this, these global keywords are documented here only and referred to within the Method Keywords subsections of the *Optional Method of Integrations* section of this routine.

Double — If present and nonzero, double precision is used.

Err_Abs — The absolute accuracy desired. (Default: *Err_Abs* = $\varepsilon^{1/2}$, where ε is the machine precision)

Err_Est — An estimate of the absolute value of the error.

Err_Rel — The relative accuracy desired. (Default: *Err_Rel* = $\varepsilon^{1/2}$, where ε is the machine precision)

Max_Subinter — The number of subintervals allowed.
(Default: *Max_Subinter* = 500)

N_Evals — A named variable into which the number of evaluations of f is stored.

N_Subinter — A named variable into which the number of subintervals generated is stored.

Discussion

In the default case, the three input parameters f, a and b are required. Additional methods of integration are also possible using INTFCN. If another method of integration is desired, a combination of these parameters, along with additional parameters and keywords must be specified. Descriptions of the additional parameters and keywords to use for specific methods of integration are discussed in the following sections.

Optional Methods of Integration

Using different combinations of keywords and parameters, one of several types of integration can be performed including:

Integration of functions based on Gauss-Kronrod rules

Integration of functions with singular points given

Integration of functions with algebraic-logarithmic singularities

Integration of functions over an infinite or semi-infinite interval

Integration of functions containing a sine or cosine factor

Computation of Fourier sine and cosine transforms

Integrals in the Cauchy principle value sense

Integration of smooth functions using a non-adaptive rule

Computation of two-dimensional iterated integrals

By specifying different sets of parameters and/or keywords, a number of different types of integration can be performed. Internally, the method to be used is determined by examining the combination of parameters and/or keywords used in the call to INTFCN. To specify a specific method of integration, refer to the appropriate discussion.

Integration of Functions Based on Gauss-Kronrod Rules

This method integrates functions using a globally adaptive scheme based on Gauss-Kronrod rules shown in the table that follows.

Synopsis

Triggered by the use of keyword *Rule*.

result = INTFCN(*f*, *a*, *b*, *Rule* = *rule*)

Returned Value

result — The value of

$$\int_a^b f(x)dx.$$

If no value can be computed, NaN is returned.

Keywords

In addition to the keywords listed for this function, the following keyword is also available:

Rule — If specified, the integral is computed using a globally adaptive scheme based on Gauss-Kronrod rules
Choice of Quadrature Rule

Rule	**Gauss-Kronrod Rule**
1	7-15 points
2	10-21 points
3	15-31 points
4	20-41 points
5	25-51 points

Choice of Quadrature Rule (Continued)

Rule	**Gauss-Kronrod Rule**
6	30-61 points

Integration of Functions with Singular Points Given

This method integrates functions with given singularity points.

Synopsis

Requires the use of keyword *Sing_Pts*.

result = INTFCN(*f*, *a*, *b*, *Sing_Pts* = *points*)

Returned Value

result — The value of

$$\int_a^b f(x)dx.$$

If no value can be computed, NaN is returned.

Keywords

In addition to the keywords listed in the *Global Keywords* section, the following keyword is available:

Sing_Pts — If present, specifies the abscissas of the singularities. These values should be interior to the interval [*a*, *b*].

Integration of Functions with Algebraic-logarithmic Singularities

This method integrates functions with algebraic-logarithmic singularities.

Input Parameters

alpha — The strength of the singularity at *a*. Must be greater than –1.

beta — The strength of the singularity at *b*. Must be greater than –1.

Synopsis

Triggered by the use of the parameters *alpha* and *beta* and one of the following keywords, in addition to *f*, *a*, and *b.*

result = INTFCN(*f*, *a*, *b*, *alpha*, *beta*, /*Algebraic*)

result = INTFCN(*f*, *a*, *b*, *alpha*, *beta*, /*Alg_Left_Log*)

result = INTFCN(*f*, *a*, *b*, *alpha*, *beta*, /*Alg_Log*)

result = INTFCN(*f*, *a*, *b*, *alpha*, *beta*, /*Alg_Right_Log*)

Returned Value

result — The value of

$$\int_a^b f(x)w(x)dx,$$

where $w(x)$ is defined by one of the following keywords. If no value can be computed, NaN is returned.

Keywords

In addition to the keywords listed in the *Global Keywords* section, the following keywords are available. Exactly one of the following keywords must be specified:

Algebraic — If present and nonzero, uses the weight function

$$(x-a)^{\alpha}(b-x)^{\beta}.$$

This is the default weight function for this method of integration.

Alg_Left_Log — If present and nonzero, uses the weight function

$$(x-a)^{\alpha}(b-x)^{\beta}\log(x-a).$$

Alg_Log — If present and nonzero, uses the weight function

$$(x-a)^{\alpha}(b-x)^{\beta}\log(x-a)\log(x-b).$$

Alg_Right_Log — If present and nonzero, uses the weight function

$$(x - a)^{\alpha}(b - x)^{\beta}\log(x - b).$$

Integration of Functions Over an Infinite or Semi-infinite Interval

This method integrates functions over an infinite or semi-infinite interval.

Input Parameters

bound — The finite limit of integration. If either of the keywords *Inf_Bound* or *Bound_Inf* are specified, this parameter is required.

Synopsis

Triggered by the presence of the function *f*, a bound (*bound*), and one of the keywords *Inf_Inf*, *Inf_Bound*, or *Bound_Inf*.

result = INTFCN(*f*, /*Inf_Inf*)

result = INTFCN(*f*, *bound*, /*Inf_Bound*)

result = INTFCN(*f*, *bound*, /*Bound_Inf*)

Returned Value

result — The value of

$$\int_a^b f(x)dx,$$

where *a* and *b* are appropriate integration limits. If no value can be computed, NaN is returned.

Keywords

In addition to the keywords listed in the *Global Keywords* section, the following keywords are available (exactly one of the following keywords must be specified):

Bound_Inf — If present and nonzero, integrates a function over the range (*bound*, *infinity*).

Inf_Bound — If present and nonzero, integrates a function over the range (*–infinity, bound*).

Inf_Inf — If present and nonzero, integrates a function over the range (*–infinity, infinity*).

Integration of Functions Containing a Sine or Cosine Factor

This method integrates functions containing a sine or a cosine factor.

Input Parameters

omega — The frequency of the trigonometric weighting function.

Synopsis

Triggered by the use of parameter *omega* and one of the keywords *Sine* or *Cosine*, in addition to *f*, *a*, and *b*.

result = INTFCN(*f*, *a*, *b*, *omega*, /*Sine*)

result = INTFCN(*f*, *a*, *b*, *omega*, /*Cosine*)

Returned Value

result — The value of

$$\int_a^b f(x)w(\omega x)dx \ ,$$

where the weight function $w(\omega x)$ as defined by the following keywords, is returned. If no value can be computed, NaN is returned.

Keywords

In addition to the keywords listed in the *Global Keywords* section, the following keywords are available (exactly one of the following keywords must be specified):

Cosine — If present and nonzero, *cos*(ω*x*) is used for the integration weight function.

Max_Moments — A scalar expression specifying an upper bound on the number of Chebyshev moments that can be stored. Increasing (decreasing) this number may increase (decrease) execution speed and space used. (Default: *Max_Moments* = 21)

Sine — If present and nonzero, *sin*(ω*x*) is used for the integration weight function.

Computation of Fourier Sine or Cosine Transforms

This method computes Fourier sine or cosine transforms.

Input Parameters

omega — The frequency of the trigonometric weighting function.

Synopsis

Triggered by the use of parameter *omega* and one of the keywords *Sine* or *Cosine*, in addition to *f* and *a*.

result = INTFCN(*f*, *a*, *omega*, /*Sine*)

result = INTFCN(*f*, *a*, *omega*, /*Cosine*)

Returned Value

result — The value of

$$\int_a^\infty f(x)w(\omega x)dx \ ,$$

where the weight function *w*(ω*x*) as defined by the following keywords, is returned. If no value can be computed, NaN is returned.

Keywords

In addition to the keywords listed in the *Global Keywords* section, the following keywords are available (exactly one of the keywords *Sine* or *Cosine* must be specified):

Cosine — If present and nonzero, *cos*(ω*x*) is used for the integration weight function.

Max_Moments — The number of subintervals allowed in the partition of each cycle. (Default: *Max_Moments* = 21)

N_Cycles — A named variable into which the number of cycles generated is stored.

Sine — If present and nonzero, *sin*(ω*x*) is used for the integration weight function.

Integrals in the Cauchy Principle Value Sense

This method computes integrals of the form

$$\int_a^b \frac{f(x)}{x-c}dx$$

in the Cauchy principal value sense.

Input Parameters

c — The singular point must not equal *a* or *b.*

Synopsis

Triggered by the use of parameter *c* and keyword *Cauchy,* in addition to *f, a*, and *b.*

result = INTFCN*(f, a*, *b, c*, /*Cauchy*)

Returned Value

result — The value of

$$\int_a^b \frac{f(x)}{x-c}dx\,.$$

If no value can be computed, NaN is returned.

Keywords

In addition to the keywords listed in the *Global Keywords* section, the following keyword is available (requires the use of keyword *Cauchy*):

Cauchy — If present and nonzero, computes integrals of the form

$$\int_a^b \frac{f(x)}{x-c}dx$$

in the Cauchy principal value sense.

Integration of Smooth Functions Using a Nonadaptive Rule

This method integrates smooth functions using a nonadaptive rule.

Synopsis

Triggered by the use of keyword *Smooth*, in addition to *f*, *a*, and *b.*

result = INTFCN(*f*, *a*, *b*, /*Smooth*)

Returned Value

result — The value of

$$\int f(x)dx.$$

If no value can be computed, NaN is returned.

Keywords

NOTE Because this method is nonadaptive, there are fewer options with the algorithm. For this method, none of the keywords described in the *Global Keywords* section apply. A complete list of the available keywords is given as follows. This method requires the use of the keyword *Smooth*.

Double — If present and nonzero, uses double precision.

Err_Abs — The absolute accuracy desired.
(Default: *Err_Abs* = $\varepsilon^{1/2}$, where ε is the machine precision)

Err_Est — A named variable into which an estimate of the absolute value of the error is stored.

Err_Rel — The relative accuracy desired.
(Default: *Err_Rel* = $\varepsilon^{1/2}$, where ε is the machine precision)

Smooth — If present and nonzero, uses a nonadaptive rule to compute the integral.

Integration of Two-dimensional Iterated Integrals

This method integrates two-dimensional iterated integrals.

Input Parameters

f — A scalar string specifying the name of a user-supplied PV-WAVE function to be integrated. Function *f* accepts two scalar parameters and returns a single scalar of the same type.

a — A scalar expression specifying the lower limit of the outer integral.

b — A scalar expression specifying the upper limit of the outer integral.

h — The name of a user-supplied PV-WAVE function used to evaluate the lower limit of the inner integral. Function *h* accepts one scalar parameter and returns a single scalar of the same type.

g — The name of a user-supplied PV-WAVE function used to evaluate the upper limit of the inner integral. Function *g* accepts one scalar parameter and returns a single scalar of the same type.

Synopsis

Triggered by the use of the parameters *g* and *h* and keyword *Two_Dimensional* in addition to *f*, *a*, and *b*.

result = INTFCN(*f*, *a*, *b*, *g*, *h*, /*Two_Dimensional*)

Returned Value

result — The value of

$$\int \int_{g(x)}^{h(x)} f(x, y)dydx.$$

If no value can be computed, NaN is returned.

Keywords

In addition to the keywords listed in the *Global Keywords* section, the following keyword is available and must be specified for this method:

Two_Dimensional — If present and nonzero, integrates a two-dimensional iterated integral.

JURYRC Procedure

Synthesizes a Cholesky-factored Toeplitz form from a stable polynomial using the Jury (reflection coefficient) algorithm.

Usage

JURYRC, *alpha_in*, *a*, *alpha_out*, *t*, *c*

Input Parameters

alpha_in — A scaling factor (or prediction error variance).

a — An array of polynomial coefficients.

Output Parameters

alpha_out — An array consisting of the elements of the diagonal matrix factor.

t — The upper triangular matrix factor.

c — An array of reflection coefficients.

Keywords

None.

Discussion

Given the coefficients of a stable, monic polynomial

$$A(z) = 1 + a_1 z^{-1} + \ldots + a_n z^{-n} ,$$

and a scaling factor (prediction error variance α), JURYRC finds an upper triangular matrix T and a diagonal matrix D that satisfy

$$aT^{-1}DT^{-T} = [\alpha, 0, 0, \ldots, 0] ,$$

where the row array a contains the coefficients of the polynomial $a = [1, a_1, \ldots, a_n]$, and the diagonal matrix is given by $D = \mathrm{diag}\ \{\alpha(n), \alpha(n-1), \ldots, \alpha(0)\}$. The matrix $T^{-1}DT^{-T} = R$ is Toeplitz. The reflection coefficients of the polynomial are also returned.

This procedure is one part of a suite of functions (JURYRC, LEVCORR, LEVDURB, and TOEPSOL) used to solve Toeplitz linear equations and factorization problems. Given the first row of a symmetric Toeplitz matrix R, the function TOEPSOL is used to solve the equation

$$Ra = \begin{bmatrix} \alpha_0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \qquad \text{(EQ 1)}$$

where α is chosen so that $a(0) = 1$ in the array a. JURYRC combined with LEVCORR are, in essence, the inverse of TOEPSOL, in that given the array of polynomial coefficients a and scalar α, EQ 1 is solved for the elements of the first row of R. This inverse operation is accomplished by first finding the elements of the matrices T and D defined above using JURYRC, and then evaluating the product
$T^{-1}DT^{-T}$ using LEVCORR.

NOTE If the zero-order coefficient, a_0, of the input polynomial is not equal to 1.0, the value of *alpha_in* is modified as *alpha_in*/a_0 and the array of polynomial coefficients is modified as a/a_0.

Example

This example illustrates the relationship between JURYRC, TOEPSOL, and LEVCORR.

```
r = [1.0d0, 0.9d0, 0.9d0^2, 0.9d0^3, 0.9d0^4, 0.9d0^5]
PM, TRANSPOSE(r), Title = $
    'First Row of Original Toeplitz Matrix'
    First Row of Original Toeplitz Matrix
    1.0000000      0.90000000      0.81000000
    0.72900000     0.65610000      0.59049000
        ; First order autoregressive autocorrelation sequence used to
        ; form first row of Toeplitz matrix.
b = [1.0d0, 0.0d0, 0.0d0, 0.0d0, 0.0d0, 0.0d0]
a = TOEPSOL(r, b)
    ; Normalize the polynomial coefficient array a so a(0) = 1.
    ; Normalization factor is alpha.
alpha = 1/a(0)
a = a/a(0)
JURYRC, alpha, a, alpha_out, t, c
    ; Compute the Cholesky decomposition (T ⁻¹D T ⁻ᵀ) from the
```

```
      ; normalized polynomial coefficients contained in the array a
      ; obtained from TOEPSOL.
LEVCORR, r_out, alpha_out, t
      ; Compute the first row of the Toeplitz matrix R = T^-1 D T^-T from
      ; the matrix and array returned from JURYRC.
PM, TRANSPOSE(r_out), $
      Title = 'First Row of Returned Toeplitz Matrix'
      First Row of Returned Toeplitz Matrix
      1.0000000      0.90000000      0.81000000
      0.72900000    0.65610000      0.59049000
```

See Also

LEVCORR, LEVDURB, TOEPSOL

For Additional Information

Proakis and Manolakis, 1992.

Roberts and Mullis, 1987, p. 527.

Many applications that involve the use of JURYRC, LEVCORR, LEVDURB, and TOEPSOL may be found in the statistical signal processing texts listed in *Background Reading* on page 29.

LEVCORR Procedure

Computes the first row (autocorrelation sequence) of a Toeplitz matrix from its Cholesky-factored form.

Usage

LEVCORR, *r*, *alpha*, *t*

Input Parameters

alpha — An array consisting of the elements of the diagonal matrix factor.

t — The upper triangular matrix factor.

Output Parameters

r — An array consisting of the first row of a Toeplitz matrix (the autocorrelation sequence).

Keywords

None.

Discussion

Given the Cholesky-factored form of a Toeplitz matrix consisting of the upper triangular matrix T and the diagonal matrix D, LEVCORR computes the first row of the Toeplitz matrix R defined by

$$T^{-1}DT^{-T} = R ,$$

where $D = \mathrm{diag}\ \{a(n), a(n-1), ..., a(0)\}$.

LEVCORR is one of a suite of functions (including JURYRC, LEVCORR, LEVDURB, and TOEPSOL) used to solve Toeplitz linear equations and factorization problems. For examples of the use of LEVCORR, see JURYRC and LEVDURB.

See Also

JURYRC, LEVDURB, TOEPSOL

For Additional Information

Proakis and Manolakis, 1992.

Roberts and Mullis, 1987, p. 527.

LEVDURB Procedure

Cholesky-factors symmetric, positive, definite Toeplitz matrices using the Levinson-Durbin algorithm.

Usage

LEVDURB, *r*, *alpha*, *t*, *c*

Input Parameters

r — An array consisting of the first row of the Toeplitz matrix.

Output Parameters

alpha — An array consisting of the elements of the diagonal matrix factor.

t — The upper triangular matrix factor, also known as the Cholesky decomposition matrix.

c — An array of reflection coefficients.

Keywords

None.

Discussion

Given the first row of the Toeplitz matrix R, the procedure finds an upper triangular matrix T and a diagonal matrix
$D = \text{diag}\{a(n), a(n-1), ..., a(0)\}$ that satisfy the following equation.

$$D = TRT^T$$

The so-called reflection coefficients of the Toeplitz matrix are also returned. LEVDURB is one of a suite of functions (including JURYRC, LEVCORR, LEVDURB, and TOEPSOL) used to solve Toeplitz linear equations and factorization problems.

LEVDURB performs the inverse operation of LEVCORR by accepting the elements of the matrices D and T and then computing the first row of the Toeplitz matrix $R = T^{-1}DT^{-T}$.

Example

This example illustrates the inverse relationship between LEVDURB and LEVCORR.

```
r = [1.0d0, 0.9d0, 0.9d0^2, 0.9d0^3, 0.9d0^4, 0.9d0^5]
```

; First order autoregressive autocorrelation sequence used
; to form the first row of Toeplitz matrix.

```
PM, TRANSPOSE(r), Title = $
    'First Row of Original Toeplitz Matrix'
    First Row of Original Toeplitz Matrix
    1.0000000      0.90000000      0.81000000
    0.72900000    0.65610000      0.59049000
LEVDURB, r, alpha, t, c
```

; Compute the Cholesky decomposition $T^{-1}DT^{-T}$ of
; the Toeplitz matrix.

```
LEVCORR, r_out, alpha, t
```

; Compute the first row of the Toeplitz matrix $R = T^{-1}DT^{-T}$ from
; the Cholesky decomposition.

```
PM, TRANSPOSE(r_out), $
    Title = 'First Row of Returned Toeplitz Matrix'
    First Row of Returned Toeplitz Matrix
    1.0000000      0.90000000      0.81000000
    0.72900000     0.65610000      0.59049000
```

See Also

JURYRC, LEVCORR, TOEPSOL

For Additional Information

Proakis and Manolakis, 1992.

Roberts and Mullis, 1987 p. 522.

LINPROG Function

Solves a linear programming problem using the revised simplex algorithm.

Usage

result = LINPROG(*a*, *b*, *c*)

Input Parameters

a — A two-dimensional matrix containing the coefficients of the constraints. The coefficient for the *i*-th constraint is contained in *A* (*i*, *).

b — A one-dimensional matrix containing the right-hand side of the constraints. If there are limits on both sides of the constraints, *b* contains the lower limit of the constraints.

c — A one-dimensional array containing the coefficients of the objective function.

Returned Value

result —The solution *x* of the linear programming problem.

Keywords

Bu — An array with *N_ELEMENTS*(*b*) elements containing the upper limit of the constraints that have both the lower and the upper bounds. If no such constraint exists, *Bu* is not needed.

Double — If present and nonzero, double precision is used.

Dual — The name of the variable into which the array with *N_ELEMENTS*(*c*) elements, containing the dual solution, is stored.

Irtype — An array with *N_ELEMENTS*(*b*) elements indicating the types of general constraints in the matrix *A*. Let

$$r_i = A_{i0}x_0 + \ldots + A_{in-1}x_{n-1} .$$

The value of *Irtype* (*i*) is described in the following table.

Values of *Irtype*

Irtype (i)	**Constraints**
0	$r_i = B_i$
1	$r_i \le B_u$
2	$r_i \ge B_i$
3	$B_i \le r_i \le B_u$

(Default: *Irtype*(*) = 0)

Itmax — The maximum number of iterations. (Default: *Itmax* = 10,000)

Obj — The name of the variable into which the optimal value of the objective function is stored.

Xlb — Array with *N_ELEMENTS*(*c*) elements containing the lower bound on the variables. If there is no lower bound on a variable, 10^{30} should be set as the lower bound. (Default: *Xlb*(*) = 0)

Xub — Array with *N_ELEMENTS*(*c*) elements containing the upper bound on the variables. If there is no upper bound on a variable, -10^{30} should be set as the upper bound. (Default: *Xub*(*) = *infinity*)

Discussion

LINPROG uses a revised simplex method to solve linear programming problems; i.e., problems of the form

$$\min_{x \in \mathbf{R}^n} c^T x$$

subject to

$$b_l \le A_x \le b_u$$

$$x_l \le x \le x_u$$

where *c* is the objective coefficient array, *A* is the coefficient matrix, and the arrays b_l, b_u, x_l, and x_u are the lower and upper bounds on the constraints and the variables.

For a complete discussion of the revised simplex method, see Murtagh (1981) or Murty (1983).

Example

The linear programming problem in the standard form

$$\min f(x) = -x_0 - 3x_1$$

and subject to

$$\begin{aligned} x_0 + x_1 + x_2 \quad &= 1.5 \\ x_0 + x_1 + \quad - x_3 \quad &= 0.5 \\ x_0 \quad + x_4 \quad &= 1.0 \\ x_1 \quad + x_5 &= 1.0 \end{aligned}$$

$$x_i \geq 0, \text{ for } i = 0, \ldots, 5$$

is solved.

```
RM, a, 4, 6
   row 0: 1 1 1  0 0 0
   row 1: 1 1 0 -1 0 0
   row 2: 1 0 0  0 1 0
   row 3: 0 1 0  0 0 1
      ; Define the coefficients of the constraints.
RM, b, 4, 1
   row 0: 1.5
   row 1:  .5
   row 2: 1
   row 3: 1
      ; Define the right-hand side of the constraints.
RM, c, 6, 1
   row 0: -1
   row 1: -3
   row 2:  0
   row 3:  0
   row 4:  0
   row 5:  0
      ; Define the coefficients of the objective function.
PM, LINPROG(a, b, c), Title = 'Solution'
      Solution
```

```
0.50000
1.00000
0.00000
1.00000
0.50000
0.00000
; Call LINPROG and print the solution.
```

For Additional Information

Murtagh, 1981.

Murty, 1983.

LPC Function

Computes the linear prediction coefficients of the denominator polynomial in a filter structure. This technique is also called the autocorrelation method of autoregressive (AR) signal modeling and the maximum entropy method (MEM) of spectrum analysis.

Usage

result = LPC(*x*, *n*)

Input Parameters

x — An array containing the data sequence to be modeled using an all-pole prediction filter.

n — The prediction filter order.

Returned Value

result — A filter structure containing the coefficients of the all-pole filter model.

Keywords

None.

Discussion

Linear prediction is a method of modeling the current value in a sequence $x(k)$ as a linear combination of the previous values in the sequence

$$x(k) \approx \hat{x}(k) = -\sum_{n=1}^{N} a_n x(k-n).$$

LPC computes the coefficients a_n, $n = 1, 2, ..., N$ to minimize the error given by

$$e(k) = x(k) - \hat{x}(k)$$

using least squares techniques. In matrix form, the problem is to minimize the least squares error of the system of equations given by

$$\begin{bmatrix} 0 & 0 & \cdots & 0 \\ x(0) & 0 & & 0 \\ x(1) & x(0) & \ddots & \vdots \\ \vdots & x(1) & & 0 \\ \vdots & \vdots & \ddots & x(0) \\ x(L) & & & x(1) \\ 0 & x(L) & \ddots & \vdots \\ \vdots & 0 & & \\ 0 & 0 & \ddots & x(L) \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_N \end{bmatrix} = \begin{bmatrix} -x(0) \\ -x(1) \\ \vdots \\ -x(L) \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

where $L + 1$ is the length of the signal sequence.

The solution to this problem is obtained by solving the Toeplitz linear equation

$$\begin{bmatrix} \hat{r}(0) & \hat{r}(1) & & \hat{r}(N-1) \\ \hat{r}(1) & \ddots & \ddots & \\ & \ddots & & \hat{r}(1) \\ \hat{r}(N-1) & & \hat{r}(1) & \hat{r}(0) \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_N \end{bmatrix} = \begin{bmatrix} -\hat{r}(1) \\ -\hat{r}(2) \\ \vdots \\ -\hat{r}(n) \end{bmatrix}$$

where the sample autocorrelation sequence is given by

$$\hat{r}(k) = \sum_{n} x(n)x(n+k).$$

LPC uses TOEPSOL to efficiently solve the Toeplitz linear equation.

The linear prediction coefficients are returned in a stable filter structure as an all-pole IIR filter given by

$$(z) = \frac{B(z)}{A(z)} = \frac{1}{1 + a_1 z^{-1} + \ldots + a_N z^{-N}}.$$

Example

This example illustrates how LPC may be used to model a first order autoregressive signal.

```
har = FILTSTR(1, [1, -.9])
RANDOMOPT, set = 29
x = RANDOM(1000, /Normal)
y = FILTER(har, x)
```

; Generate a first order autoregressive signal.

```
h = LPC(y, 5)
```

; Compute the coefficients of the all-pole filter model

```
hf = FREQRESP_Z(h, Outfreq = f)
PLOT, f, ABS(hf), Linestyle = 2, Title = $
   'Signal Spectrum (solid line) and AR(5) ' + $
   'Model Spectrum', $
   XTitle = 'Normalized Frequency'
harf = FREQRESP_Z(har, Outfreq = f)
OPLOT, f, ABS(harf)
```

; Plot the signal spectrum and model spectrum (*Figure 2-26*).

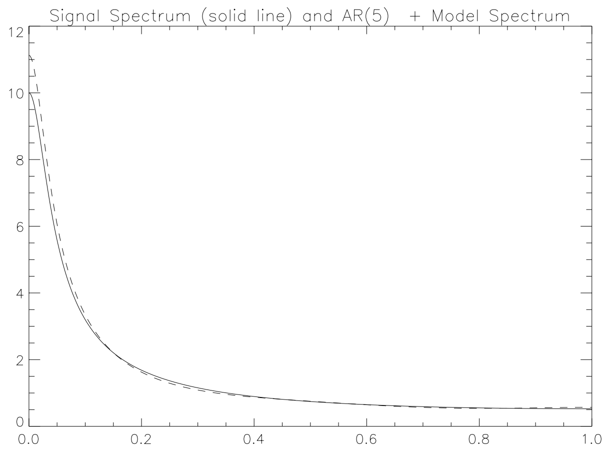


Figure 2-26 Plot of the signal spectrum and the autoregressive prediction signal using LPC.

See Also

TOEPSOL

For Additional Information

Jayant and Noll 1984, pp. 267-269.

Scharf 1991, pp. 452-456.

Therrien 1992, pp. 536-537.

NLINLSQ Function

Solves a nonlinear least-squares problem using a modified Levenberg-Marquardt algorithm.

Usage

result = NLINLSQ(*f*, *m*, *n*)

Input Parameters

f — A scalar string specifying a user-supplied function to evaluate the function that defines the least-squares problem. The *f* function accepts the following two parameters and returns an array of length *m* containing the function values at *x*:

> ***m*** — The number of functions.
>
> ***x*** — An array of length *n* containing the point at which the function is evaluated.

m — The number of functions.

n — The number of variables where $n \le m$.

Returned Value

result —The solution *x* of the nonlinear least-squares problem.

Keywords

Double — If present and nonzero, double precision is used.

Fjac — The name of the variable into which an array of size *n*-by-*m* containing the Jacobian at the approximate solution is stored.

FScale — An array with *m* components containing the diagonal scaling matrix for the functions. The ***i***-th component of *FScale* is a positive scalar specifying the reciprocal magnitude of the *i*-th component function of the problem. (Default: *FScale*(*) = 1)

Fvec — The name of the variable into which a real array of length *m* containing the residuals at the approximate solution is stored.

Good_Digits — The number of good digits in the function. (Default: machine dependent)

Intern_Scale — An internal variable scaling option. With this keyword, the values for *XScale* are set internally.

Itmax — The maximum number of iterations. (Default: *Itmax* = 100)

Jacobian — A scalar string specifying a user-supplied function to compute the Jacobian. This function accepts two parameters and returns an *n*-by-*m* array containing the Jacobian at the input point *s*. Note that each derivative $\partial f_i / \partial x_i$ should be returned in the (*i*, *j*) element of the returned matrix. The parameters of the function are as follows:

> ***m*** — The number of equations.
>
> ***x*** — An array of length *n* at which the point Jacobian is evaluated.

JTJ_inverse — The name of the variable into which an array of size *n*-by-*m* containing the inverse matrix of J^TJ, where *J* is the final Jacobian, is stored. If J^TJ is singular, the inverse is a symmetric g_z inverse of J^TJ. (See the *PV-WAVE: IMSL Mathematics Reference* CHNNDSOL function for a discussion of generalized inverses and the definition of the g_z inverse.)

Max_Evals — The maximum number of function evaluations. (Default: *Max_Evals* = 400)

Max_Jacobian — The maximum number of Jacobian evaluations. (Default: *Max_Jacobian* = 400)

Max_Step — The maximum allowable step size.
(Default: *Max_Step* = 1000max(ε_1, ε_2),
where

$$\varepsilon_1 = \sqrt{\sum_{i=1}^{n} s_i t_i^{\,2}}, \text{ and } \varepsilon_2 = \|s\|_2,$$

s = *XScale*, and t = *XGuess*)

Rank — The name of the variable into which the rank of the Jacobian is stored.

Tol_Afcn — The absolute function tolerance. (Default: *Tol_Afcn* = max(10^{-20}, ε^2), [max(10^{-40}, ε^2) in double], where ε is the machine precision)

Tolerance — The tolerance used in determining linear dependence for the computation of the inverse of J^TJ. (If *Jacobian* is specified, *Tolerance* = 100ε where ε is the machine precision, is the default; otherwise, $\varepsilon^{1/2}$, where ε is the machine precision, is the default.)

Tol_Grad — The scaled gradient tolerance. The *i*-th component of the scaled gradient at *x* is calculated as

$$\frac{|g_i| \times \max(|x_i|, 1/s_i)}{\frac{1}{2}\|F(x)\|_2^2},$$

where $g = \nabla F(x)$, $s = XScale$, and

$$\|F(x)\|_2^2 = \sum_{i=1}^{m} f_i(x)^2.$$

(Default: *Tol_Step* = $\varepsilon^{1/2}$ ($\varepsilon^{1/3}$ in double), where ε is the machine precision)

Tol_Rfcn — The relative function tolerance. (Default: *Tol_Rfcn* = $\max(10^{-10}, \varepsilon^{2/3})$, [$\max(10^{-40}, \varepsilon^{2/3})$ in double], where ε is the machine precision)

Tol_Step — The scaled step tolerance. The *i*-th component of the scaled step between two points *x* and *y* is computed as

$$\frac{|x_i - y_y|}{\max(|x_i|, 1/s_i)}$$

where s = *XScale*. (Default: *Tol_Step* = $\varepsilon^{2/3}$, where ε is the machine precision)

Trust_Region — The size of initial trust-region radius. (Default: based on the initial scaled Cauchy step)

XGuess — An array with N components containing an initial guess. (Default: *XGuess*(*) = 0)

XScale — An array with *n* components containing the scaling array for the variables. Keyword *XScale* is used mainly in scaling the gradient and the distance between two points. See keywords *Tol_Grad* and *Tol_Step* for more detail. (Default: *XScale*(*) = 1)

See CHNNDSOL for a discussion of generalized inverses and the definition of the g_z inverse.

Discussion

NLINLSQ is based on the MINPACK routine LMDER by Moré et al. (1980), and uses a modified Levenberg-Marquardt method to solve nonlinear least-squares problems.

(For more details about the algorithms used by NLINLSQ, refer to the description of NLINLSQ in the *PV-WAVE: IMSL Mathematics Reference*.)

The problem is stated as follows:

$$\text{in } \frac{1}{2}F(x)^T F(x) = \frac{1}{2}\sum_{i=1}^{m} f_i(x)^2,$$

where $m \geq n$, $F : \mathrm{R}^n \rightarrow \mathrm{R}^m$ and $f_i(x)$ is the i-th component function of $F(x)$. From a current point, the algorithm uses the trust region approach

$$\min_{x \in \mathbf{R}^n} \|F(x_c) + J(x_c)(x_n - x_c)\|_2$$

subject to $\|x_n - x_c\|_2 \leq \delta_c$

to get a new point x_0, compute x_s as

$$x_n = x_c - [\, J(x_c)^T J(x_c) + \mu_c I]^{-1} J(x_c)^T F(x_c)$$

where $\mu_c = 0$ if $\delta_c \geq \left\| (J(x_c)^T J(x_c))^{-1} J(x_c)^T F(x_c) \right\|_2$,

and $\mu_c \geq 0$ otherwise. The value μ_c is defined by the function. The array and matrix $F(x_c)$ and $J(x_c)$ are the function values and the Jacobian evaluated at the current point x_c. This function is repeated until the stopping criteria are satisfied.

The first stopping criterion for NLINLSQ occurs when the norm of the function is less than the absolute function tolerance, *Tol_Afcn*. The second stopping criterion occurs when the norm of the scaled gradient is less than the given gradient tolerance *Tol_Grad*. The third stopping criterion for NLINLSQ occurs when the scaled distance between the last two steps is less than the step tolerance *Tol_Step*. For more details, see Levenberg (1944), Marquardt (1963), or Dennis and Schnabel (1983, Chapter 10).

Example

This example uses the nonlinear data-fitting problem found in Dennis and Schnabel (1983, p. 225).

NOTE For more information on the problem that is being solved in this example, refer to the description of NLINLSQ in the *PV-WAVE: IMSL Mathematics Reference*.

The problem is stated as follows:

$$\text{in}\frac{1}{2}\sum_{i=0}^{3} f_i(x)^2$$

where

$$f_i(x) = e^{t_i x} - y_i \ ,$$

is solved with the data t = [1, 2, 3] and y = [2, 4, 3].

```
.RUN
   ; Define the function that defines the least-squares problem.
FUNCTION f, m, x
y = [2, 4, 3]
t = [1, 2, 3]
RETURN, EXP(x(0) * t) - y
END
   % Compiled module: F.
solution = NLINLSQ("f", 3, 1)
PM, solution, Title = 'The solution is:'
   The solution is:
   0.440066
      ; Output the results.
PM, f(m, solution), Title = $
   'The function values are:'
   The function values are:
   -0.447191
   -1.58878
    0.744159
```

See Also

In the *PV-WAVE: IMSL Mathematics Reference*:

CHNNDSOL

For Additional Information

Dennis and Schnabel, 1983, Chapter 10.

Levenberg, 1944.

Marquardt, 1963.

Moré, Garbow and Hillstrom,1980.

NONLINPROG Function

Solves a general nonlinear programming problem using the successive quadratic programming (QP) algorithm.

Usage

result = NONLINPROG(*f*, *m*, *n*)

Input Parameters

f — A scalar string specifying a user-supplied procedure to evaluate the function at a given point. Procedure *f* has the following parameters:

> ***m*** — The total number of constraints.
>
> ***meq*** — The number of equality constraints.
>
> ***x*** — A one-dimensional array at which point the function is evaluated.
>
> ***active*** — A one-dimensional array with *mmax* components indicating the active constraints where *mmax* is the maximum of (1, *m*).
>
> ***f*** — The computed function value at the point *x*. (Output)
>
> ***g*** — A one-dimensional array with *mmax* components containing the values of the constraints at point *x*, where *mmax* is the maximum (1, *m*). (Output)

m — The total number of constraints.

n — The number of variables.

Returned Value

result —The solution of the nonlinear programming problem.

Keywords

Double — If present and nonzero, double precision is used.

Err_Rel — The final accuracy. (Default: *Err_Rel* = $\varepsilon^{1/2}$, where ε is the machine precision)

Grad — A scalar string specifying a user-supplied procedure to evaluate the gradients at a given point. The procedure specified by *Grad* has the following parameters:

> ***mmax*** — The maximum of (1, *m*).
>
> ***m*** — The total number of constraints.
>
> ***meg*** — The number of equality constraints.
>
> ***x*** — The array at which point the function is evaluated.
>
> ***active*** — An array with *mmax* components indicating the active constraints.
>
> ***f*** — The computed function value at the point *x*.
>
> ***g*** — An array with *mmax* components containing the values of the constraints at point *x*.
>
> ***df*** — An array with *n* components containing the values of the gradient of the objective function. (Output)
>
> ***dg*** — An array of size *n*-by-*mmax* containing the values of the gradients for the active constraints. (Output)

Ibtype — A scalar indicating the types of bounds on the variables as shown in the following table.

Ibtype Values for Variable Bounds

Ibtype	**Action**
0	User supplies all the bounds.
1	All variables are nonnegative.

Ibtype Values for Variable Bounds

Ibtype	**Action**
2	All variables are non-positive.
3	User supplies only the bounds on first variable; all other variables have the same bounds.

(Default: no bounds are enforced)

Itmax — The maximum number of iterations allowed. (Default: *Itmax* = 200)

Meq — The number of equality constraints. (Default: *Meg* = *m*)

Obj — The name of a variable into which a scalar containing the value of the objective function at the computed solution is stored.

XGuess — An array with *n* components containing an initial guess of the computed solution. (Default: *XGuess*(*) = 0)

Xlb — A named variable, containing a one-dimensional array with *n* components, containing the lower bounds on the variables. (Input, if *Ibtype* = 0; Output, if *Ibtype* = 1 or 2; Input/Output, if *Ibtype* = 3). If there is no lower bound on a variable, the corresponding *Xlb* value should be set to -10^6 . (Default: no lower bounds are enforced on the variables)

XScale — An array with *n* components containing the reciprocal magnitude of each variable. Keyword *XScale* is used to choose the finite-difference step size, *h*. The ***i***-th component of *h* is computed as
$\varepsilon^{1/2}$*max($|x_i|$, $1/s_i$) *sign(x_i), where ε is the machine precision,
s = *XScale*, and *sign*(x_i) = 1 if $x_i \geq 0$; otherwise, *sign*(x_i) = –1.
(Default: *XScale*(*) = 1)

Xub — A named variable, containing a one-dimensional array with *n* components, containing the upper bounds on the variables. (Input, if *Ibtype* = 0; Output, if *Ibtype* = 1 or 2; Input/Output, if *Ibtype* = 3). If there is no upper bound on a variable, the corresponding *Xub* value should be set to 10^6. (Default: no upper bounds are enforced on the variables)

Discussion

NONLINPROG is based on the subroutine NLPQL developed by Schittkowski (1986), and uses a successive quadratic programming method to solve the general nonlinear programming problem. The problem is stated as follows:

$$\min_{x \in \mathbf{R}^n} f(x)$$

subject to

$$g_j(x) = 0 \text{ , for } j = 0, \ldots, m_e - 1$$

$$g_j(x) \geq 0 \text{ , for } j = m_e, \ldots, m - 1$$

$$(x_l \leq x \leq x_u) \text{ ,}$$

where all problem functions are assumed to be continuously differentiable. The method, based on the iterative formulation and solution of quadratic programming (QP) subproblems, obtains these subproblems by using a quadratic approximation of the Lagrangian and by linearizing the constraints. That is,

$$\min_{x \in \mathbf{R}^n} \frac{1}{2} d^T B_k d + \nabla f(x_k)^T d$$

subject to

$$\nabla g_j(x_k)^T d + g_j(x_k) = 0 \text{ , for } j = 0, \ldots, m_e - 1$$

$$\nabla g_j(x_k)^T d + g_j(x_k) \geq 0 \text{ , for } j = m_e, \ldots, m - 1$$

$$x_l - x_k \leq d \leq x_u - x_k$$

where B_k is a positive definite approximation of the Hessian and x_k is the current iterate. Let d_k be the solution of the subproblem. A line search is used to find a new point x_{k+1}

$$x_{k+1} = x_k + \lambda d_k \qquad \lambda \in (0, 1]$$

such that a "merit function" has a lower function value at the new point. Here, the augmented Lagrange function (Schittkowski 1986) is used as the merit function.

When optimality is not achieved, B_k is updated according to the modified BFGS formula (Powell 1978). Note that this algorithm may generate infeasible points during the solution process. Therefore, if feasibility must be maintained for intermediate points, this function may not be suitable. For more theoretical and practical details, see Stoer (1985), Schittkowski (1983, 1986), and Gill et al. (1985).

For more details about the algorithms used by NONLINPROG, refer to the description of NONLINPROG in the *PV-WAVE: IMSL Mathematics Reference*.

Example

The problem

$$\min F(x) = (x_1 - 2)^2 + (x_2 - 1)^2$$

subject to

$$g_1(x) = x_1 - 2x_2 + 1 = 0$$

$$g_2(x) = -x_1^2/4 - x_2^2 + 1 \geq 0$$

is solved.

```
.RUN
   ; Define the procedure to evaluate the function at a given point.
PRO f, m, meq, x, active, f, g
   tmp1 = x(0) - 2.
   tmp2 = x(1) - 1.
   f = tmp1^2 + tmp2^2
   g = FLTARR(2)
   IF active(0) THEN g(0) = x(0) - 2. * x(1) + 1.
   IF active(1) THEN g(1) = -(x(0)^2)/4. - x(1)^2 + 1.
END
   % Compiled module: F.
x = NONLINPROG('f', 2, 2, Meq = 1)
   ; Compute the solution.
PM, x, Title = 'Solution:'
   Solution:
   0.822902
   0.911452
```

See Also

In the *PV-WAVE: IMSL Mathematics Reference*:

NONLINPROG

For Additional Information

Gill, Murray, Saunders and Wright, 1985.

Powell, 1978.

Schittkowski, 1983, 1986.

Stoer, 1985.

NORM Function

Computes various norms of an array, or the difference of two arrays.

Usage

result = NORM(*x* [, *y*])

Input Parameters

x — An array for which the norm is to be computed.

y — (optional) If present, NORM computes the norm of ($x - y$).

Returned Value

result — The requested norm of the input array. If the norm cannot be computed, NaN is returned.

Keywords

Index_Max — A named variable into which the index of the element of *x* with the maximum modulus is stored. If *Index_Max* is used, the keyword *Inf* must also be used. If the parameter *y* is specified, the index of ($x - y$) with the maximum modulus is stored.

Inf — If present and nonzero, computes the infinity norm max$|x_i|$.

One — If present and nonzero, computes the 1-norm

$$\sum_{i=0}^{n-1} |x_i| .$$

Discussion

For more detailed information, see the description of NORM in the *PV-WAVE Personal Edition:Statistics Toolkit Reference*.

By default, NORM computes the Euclidean norm as follows:

$$\left(\sum_{i=0}^{n-1} x_i^2 \right)^{\frac{1}{2}}$$

If the keyword *One* is set, the 1-norm

$$\sum_{i=0}^{n-1} |x_i|$$

is returned. If the keyword *Inf* is set, the infinity norm

$$\max |x_i|$$

is returned. In the case of the infinity norm, the index of the element with maximum modulus also is returned.

If the parameter *y* is specified, the computations of the norms described above are performed on $(x - y)$.

Example 1

In this example, the Euclidean norm of an input array is computed.

```
x = [1.0, 3.0, -2.0, 4.0]
n = NORM(x)
PM, n, Title = 'Euclidean norm of x:'
   Euclidean norm of x:
      5.47723
```

Example 2

This example computes max $| x_i - y_i |$ and prints the norm and index.

```
x = [1.0, 3.0, -2.0, 4.0]
y = [4.0, 2.0, -1.0, -5.0]
n = NORM(x, y, /Inf, Index_Max = imax)
PM, n, Title = 'Infinity norm of (x-y):'
   Infinity norm of (x-y):
      9.00000
PM, imax, Title = $
   'Element of (x-y) with maximum modulus:'
   Element of (x-y) with maximum modulus:
      3
```

OPLOTCOMB Procedure

Produces a comb plot over a previously drawn plot.

Usage

OPLOTCOMB, *signal*

OPLOTCOMB, *abscissa*, *signal*

Input Parameters

abscissa — A one-dimensional array containing the abscissa of the signal to be plotted.

signal — A one-dimensional array containing the signal to be plotted.

Keywords

Color — Specifies the index of the plot color to use for the comb plot.

Discussion

OPLOTCOMB uses OPLOT to produce a comb plot over a previously drawn plot.

Example

In this example, OPLOTCOMB is used to overlay a plot on a plot previously drawn using PLOTCOMB.

```
n = 50
```

```
t = 2*!Pi*(FINDGEN(n)/n)
PLOTCOMB, t, SIN(t)
      ; Plot one full period of a sine wave, using PLOTCOMB.
!P.Linestyle = 1
      ; Change line style to dotted.
OPLOTCOMB, t, -SIN(t)
      ; Multiply the sine wave by (–1) and plot with dotted lines
      ; using the OPLOTCOMB procedure.
```

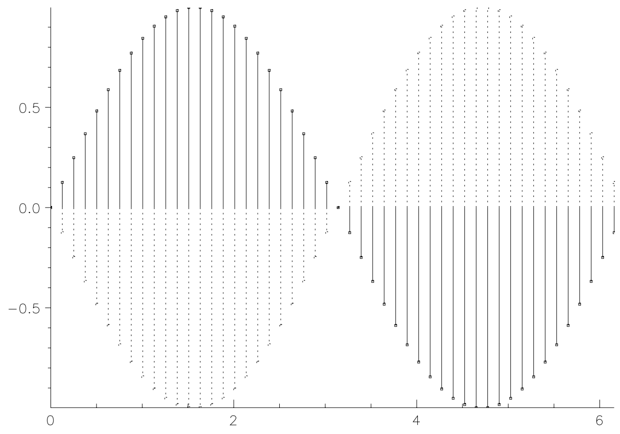


Figure 2-27 An inverted sine wave comb plot (dotted comb lines) is overlaid (using OPLOTCOMB) on top of the original sine wave comb plot (solid comb lines).

See Also

PLOTCOMB

In the *PV-WAVE Reference*:

OPLOT, PLOT

PAIRCONJ Function

Sorts a one-dimensional array of complex values into complex conjugate pairs.

Usage

result = PAIRCONJ(*z*)

Input Parameters

z — A one-dimensional array of complex values.

Returned Value

result — A complex array with the values sorted into complex conjugate pairs.

Keywords

None.

Discussion

PAIRCONJ returns a one-dimensional array containing the input data, but rearranged into complex conjugate pairs. If there are values in *z* that do not appear to match up, they are put at the end of the output array. If there are an odd number of complex values in *z* (with the imaginary part not equal to zero), the function returns 0 and an error message.

Example

In this example, PAIRCONJ is used on two sets of complex conjugate pairs. PAIRCONJ returns the data as one set of ordered complex conjugate pairs.

```
z = COMPLEX([2,1,2,1],[9,-3,-9,3])
    ; The input is a simple array of complex conjugates all mixed up.
```

```
PM, z
    (      2.00000,      9.00000)
    (      1.00000,     -3.00000)
    (      2.00000,     -9.00000)
    (      1.00000,      3.00000)
```

```
z2 = PAIRCONJ(z)
    ; Complex values are output in conjugate pairs.
```

```
PM, z2
    (      1.00000,     -3.00000)
    (      1.00000,      3.00000)
    (      2.00000,     -9.00000)
    (      2.00000,      9.00000)
```

See Also

PAIRINV

PAIRINV Function

Sorts a set of complex values into reciprocal pairs.

Usage

result = PAIRINV(*z*)

Input Parameters

z — A one-dimensional array of complex values.

Returned Value

result — A complex array with the values sorted into reciprocal pairs.

Keywords

None.

Discussion

PAIRINV returns an array of the input data sorted into reciprocal pairs. Each pair (except |*z*| = 1) must have one value inside the unit circle
(|*z*| < 1) and one outside (|*z*| > 1). The value inside the unit circle is the first element of the pair and the value outside the unit circle is the second. If there are values in *z* that do not appear to match up, they are put at the end of the output array in no particular order.

Example

In this example, the input to PAIRINV is a jumbled set of reciprocal pairs of complex numbers.

```
z = COMPLEX(1,1)
    ; A complex number z = 1 + j.
a = [z*3, 1/z, 1/(z*3), z]
    ; Create the jumbled set of reciprocal pairs.
PM, a
    (      3.00000,      3.00000)
    (     0.500000,    -0.500000)
    (     0.166667,    -0.166667)
    (      1.00000,      1.00000)
p = PAIRINV(a)
PM, p
    (     0.500000,    -0.500000)
    (      1.00000,      1.00000)
    (     0.166667,    -0.166667)
    (      3.00000,      3.00000)
```

See Also

PAIRCONJ

PARSEFILT Procedure

Parses a filter structure and returns the individual members of the structure.

Usage

PARSEFILT, *h*, *name*, *b*, *a*

Input Parameters

h — A filter structure.

Returned Value

name — A scalar string containing the name field of the filter.

b — The coefficients of the numerator polynomial of the filter.

a — The coefficients of the denominator polynomial of the filter.

Keywords

None.

Discussion

PARSEFILT parses a filter structure, $H(z)$

$$H(z) = \frac{B(z)}{A(z)} = \frac{b_0 + b_1 z^{-1} + \ldots + b_M z^{-M}}{1 + a_1 z^{-1} + \ldots + a_N z^{-N}}$$

and returns the individual coefficients of the numerator and the denominator polynomials of the structure.

Example

In this example, an IIR filter is created using IIRDESIGN, and the individual parts of the filter are extracted using PARSEFILT.

```
h = IIRDESIGN(7, .5, /Butter)
```

```
PARSEFILT, h, name, b, a
```

```
PM, name, Title = 'Filter name:'
   Filter name:
      Butterworth
PM, b, Title = 'Numerator Coefficients:'
   Numerator Coefficients:
      0.016565294
      0.11595706
      0.34787117
      0.57978528
      0.57978528
      0.34787117
      0.11595706
      0.016565294
PM, a, Title = 'Denominator Coefficients:'
   Denominator Coefficients:
      1.0000000
      0.0000000
      0.91997300
      0.0000000
      0.19270116
      0.0000000
      0.0076834509
      0.0000000
```

See Also

FILTSTR

PARSEWAVELET Function

Extracts the wavelet transform coefficients from a wavelet data structure.

Usage

result = PARSEWAVELET(*waveletstruct*, *n*)

Input Parameters

waveletstruct — The output of the forward wavelet transform created by the WAVELET function.

n — An index of the wavelet transform coefficient data structure level.

NOTE The input parameter *n* must be between 1 and the number of QMFs + 1, inclusive.

Returned Value

result — A double-precision array of coefficients of the *n*-th level of the wavelet transform.

Keywords

None.

Discussion

PARSEWAVELET extracts the coefficients from the *n*-th level of the wavelet transform from the wavelet transform data structure. For the definition of the wavelet transform levels, see WAVELET.

The index *n* passed into PARSEWAVELET is the number of the transform stage from which to retrieve the output. If *n* is one greater than the number of stages, the lowpass output of the last stage is returned instead.

For an example of the use of PARSEWAVELET, see the WAVELET Function example.

See Also

WAVELET

PLOTCOMB Procedure

Creates a comb plot of an input signal.

Usage

PLOTCOMB, *signal*

PLOTCOMB, *abscissa*, *signal*

Input Parameters

abscissa — A one-dimensional array containing the abscissa of the signal to be plotted.

signal — A one-dimensional array containing the signal to be plotted.

Keywords

Color — Specifies the index of the plot color to use for the comb plot.

Title — A string containing the title above the plot.

Xrange — A two-element array containing the desired data range for the *x*-axis.

Xtitle — A string containing the title of the *x*-axis.

Yrange — A two-element array containing the desired data range for the *y*-axis.

Ytitle — A string containing the title of the *y*-axis.

Discussion

PLOTCOMB uses PLOT and OPLOT to produce a comb plot. Keywords can be used to add a plot title, axis titles, and to specify the range of the axis to be plotted.

Example

In this example, a comb plot is created for a combined sinusoidal signal.

$$x(n) = \sin\left(\frac{2\pi n}{N}\right) + \frac{1}{2}\cos\left(\frac{6\pi n}{N}\right) \qquad \text{for n = 0, 1, ..., N = 49}$$

```
t = FINDGEN(50)/49
PLOTCOMB, t, SIN(2*!Pi*t) + .5*COS(6*!Pi*t)
```
; Generate the comb plot of the signal.

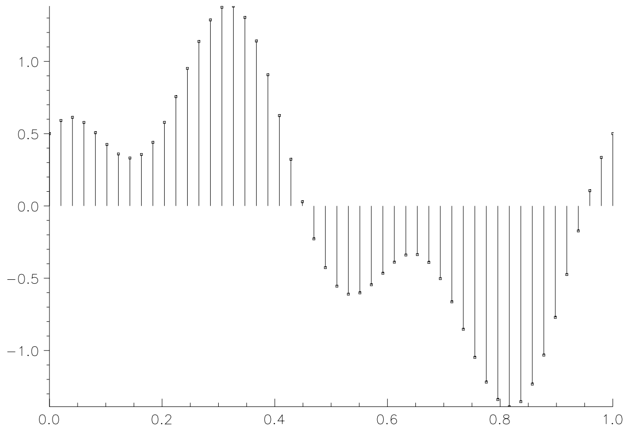


Figure 2-28 Comb plot of the combined sinusoidal signal.

See Also

OPLOTCOMB

In the *PV-WAVE Reference*:

OPLOT, PLOT

PLOTZP Procedure

Creates a root locus (zero-pole) plot.

Usage

PLOTZP, *h*

Input Parameters

h — The filter structure.

Keywords

Lines — If present and nonzero, causes lines to be drawn from the origin to the poles and zeros.

Print — If present and nonzero, causes the poles and zeros to be printed to the standard output.

Radius — If present and nonzero, overrides the automatic range calculations and sets the *x*- and *y*-ranges of the plot to the specified range [–r, r].

Title — A string containing the title of the plot.

Discussion

For a given filter structure $H(z)$ of the form

$$H(z) = \frac{B(z)}{A(z)} = \frac{b_0 + b_1 z^{-1} + \ldots + b_M z^{-M}}{a_0 + a_1 z^{-1} + \ldots + a_N z^{-N}},$$

zeros are the values of *z*, which when used to evaluate the filter structure numerator polynomial return the value zero. The poles are the values of *z* which return the value zero when used to evaluate the denominator polynomial of the filter structure.

PLOTZP plots the zeros of $H(z)$ with a circular plot symbol (“O”), and uses an “X” to indicate the poles of $H(z)$.

CAUTION It is important to note that PLOTZP uses root finding to obtain the zeros and poles of the filter structure. If either the numerator or the denominator polynomial have repeated roots, the function may generate error messages.

Example 1

In this example, the poles and zeros of an elliptic filter are plotted.

```
h = IIRDESIGN(4, .5, .05, .05, /Ellip)
```
; Create an elliptical IIR filter.

```
PLOTZP, h
```
; Plot the poles and zeros of the filter (see *Figure 2-29*).

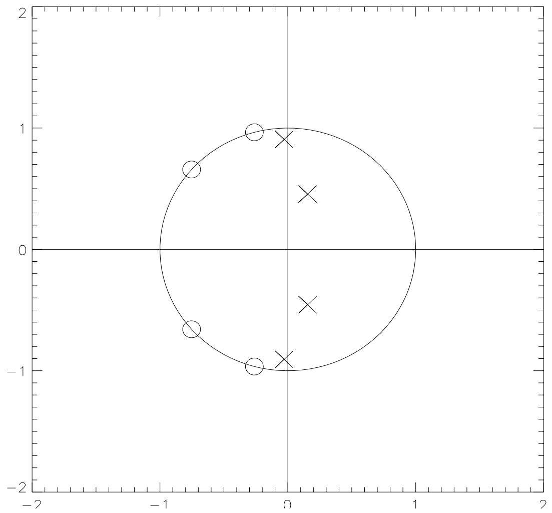


Figure 2-29 Zero-pole plot for an elliptic filter. The "O"s represent zeros of $H(z)$, and the "X"s represent the poles of $H(z)$.

Example 2

In this example, PLOTZP prints out the poles and zeros and draws lines from the origin to the poles and zeros.

```
h = IIRDESIGN(4, .5, .05, .05, /Ellip)
```
; Create an elliptical IIR filter.

```
PLOTZP, h, /Print, /Lines
   The zeros are at:
```

```
(    -0.752679,      0.658388)(    -0.752679,     -0.658388)
(    -0.261730,      0.965141)(    -0.261730,     -0.965141)
   The poles are at:
(   -0.0287310,      0.906404)(   -0.0287310,     -0.906404)
(     0.154772,      0.456417)(     0.154772,     -0.456417)
```

; Print the numerical values of the poles and zeros, plot the
; poles and zeros, and draw lines to them from the origin
; (see *Figure 2-30*).

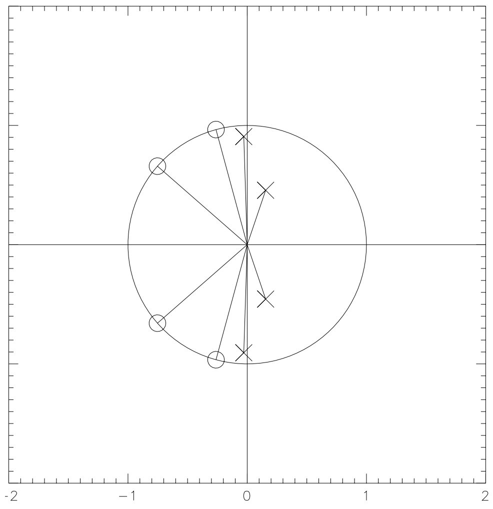


Figure 2-30 Zero-pole plot created with the *Lines* keyword specified.

See Also

ZEROPOLY

P_DEG Function

Numerically determines the degree of a polynomial.

Usage

result = P_DEG(*a*)

Input Parameters

a — An array of coefficients of a polynomial.

Returned Value

result — The degree of the polynomial.

Keywords

Epsilon — Specifies a value above which P_DEG ignores higher order coefficients of the polynomial. If *Epsilon* is not set, the default value is based on the square root of machine epsilon.

Discussion

For a given polynomial $A(z)$, such as

$$A(z) = a_0 + a_1 z^{-1} + a_2 z^{-2} + \ldots + a_N z^{-N} ,$$

P_DEG determines $M \le N$ such that $a_i \le Epsilon$ for $i = M + 1, M + 2, \ldots, N$.

Example

In this example, the degree of a polynomial
$A(z) = 5 + 4z^{-1} + 8z^{-2} + z^{-3}$ is computed with trailing zeros.

```
a= [5, 4, 8, 1, 0, 0, 0, 0]
```

```
PRINT, P_DEG(a)
    3
```

P_DIV Procedure

Divides two polynomials and returns the quotient polynomial and the remainder polynomial.

Usage

P_DIV, *b, a, q, r*

Input Parameters

b — The array of real coefficients of the numerator polynomial.

a — The array of real coefficients of the denominator polynomial.

Output Parameters

q — The array of coefficients of the quotient polynomial.

r — The array of coefficients of the remainder polynomial.

Keywords

Epsilon — Specifies a value above which the procedure ignores all higher order coefficients of the polynomial. If *Epsilon* is not set, the default value is 0.0d.

Discussion

For two polynomials $b(x)$ and $a(x)$, such that

$$b(x) = b_0 + b_1 x + b_2 x^2 + \ldots + b_M x^M \quad \text{and}$$

$$a(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_N x^N ,$$

P_DIV computes $q(z)$, called the quotient polynomial, and $r(z)$, the remainder polynomial. P_DIV uses the following relationship

$$b(x) = a(x)q(x) + r(x) .$$

Example

In this example, P_DIV is used to compute the quotient of a polynomial with real coefficients $b(x) = 2 + 7x + 6x^2 + x^3$, divided by $a(x) = 1 + x$.

$$\frac{(2 + 7x + 6x^2 + x^3)}{(1 + x)} = \frac{(1 + x)(2 + 5x + x^2)}{(1 + x)} = (2 + 5x + x^2)$$

```
 P_DIV, [2, 7, 6, 1], [1, 1], q, r
 PM, q
    2.0000000
    5.0000000
    1.0000000
        ; The quotient polynomial is q(x) = 2 + 5x + x^2.
PM, r
    0.0000000
        ; No remainder.
```

See Also

P_MULT

For Additional Information

Blahut, 1985.

P_MULT Function

Multiplies two polynomials.

Usage

result = P_MULT(*a*, *b*)

Input Parameters

a — The array of coefficients of the first polynomial.

b — The array of coefficients of the second polynomial.

Returned Value

result — An array of coefficients representing the product of the two polynomials.

Keywords

None.

Discussion

Polynomial multiplication simply amounts to convolving the coefficients of the polynomials. P_MULT uses the CONVOL1D function to perform this operation.

Example

In this example, two polynomials with real coefficients are multiplied: $a(x) = 1 + x$ and $b(x) = 1 + 2x + 2x^2 + x^3$. The solution should be
$(1 + x)(1 + 2x +2x^2 + x^3) = 1 + 3x + 4x^2 + 3x^3 + x^4$.

```
a = [1, 1]
b = [1, 2, 2, 1]
c = P_MULT(a, b)
PM, c
   1.0000000
   3.0000000
   4.0000000
   3.0000000
   1.0000000
       ; Which matches the expected solution.
```

See Also

CONVOL1D

For Additional Information

Blahut, 1985

P_SQRT Function

Polynomial spectral factorization.

Usage

result = P_SQRT(*c*)

Input Parameters

c — An array containing the coefficients of a polynomial.

Returned Value

result — An array of coefficients of the minimum phase spectral factor.

Keywords

None.

Discussion

Given a nonnegative, symmetric polynomial $c(z)$, such that

$$c(z) = \mathrm{c}(z^{-1}) > 0 \ ,$$

find another polynomial $a(z)$ such that

$$c(z) = a(z)a(z^{-1}) \ .$$

The array *c* is in the standard form for ZEROPOLY, which means that the polynomial is:

$$p = c_0 + c_1 z + c_2 z^2 + \ldots + c_n z^n \ .$$

NOTE The polynomial returned by P_SQRT is made of positive powers of *z*. To get a polynomial in z^{-1}, use the REVERSE function to reverse the coefficient array.

CAUTION If the polynomial returned from P_SQRT is multiplied by its reverse, such as you can do using the PV‑WAVE function REVERSE, the result may be scaled differently than the original array. The returned polynomial may be off by a constant factor.

Example

In this example, P_SQRT is used with the polynomial $a(x)$, such that

$$x^2 + 2x + 1 = (x + 1)^2 = a(x) .$$

```
a = [1, 2, 1]
    ; Because x^2 + 2x + 1 = (x + 1)(x + 1), factorizing a(x) should
    ; return (x + 1).
PM, P_SQRT(a)
    1.0000000
    1.0000000
```

See Also

ZEROPOLY

In the *PV-WAVE Reference*:

REVERSE

P_STAB Function

Stabilizes a filter polynomial by reflecting any roots that exist outside the unit circle to points within the unit circle.

Usage

result = P_STAB(*c*)

Input Parameters

c — An array of coefficients of a polynomial in positive powers of the independent variable, such as would be used in ZEROPOLY.

Returned Value

result — An array of polynomial coefficients of the transformed, stable polynomial.

Keywords

Information — If present and nonzero, information regarding the number of unstable roots in the polynomial is returned.

Scale — If present and nonzero, the output polynomial is scaled such that the first element is the same as the first element of the input polynomial.

Discussion

Given a polynomial whose roots lie either inside or outside of the unit circle, another polynomial is created whose roots are all inside the circle and is therefore stable. This is done by mirroring the outsiders into the circle. The transform results in a polynomial that is stable, but has the same magnitude response. The magnitude response is thus maintained, while the polynomial becomes stable.

CAUTION P_STAB relies on root finding which can create problems if the input polynomial has repeated roots. P_STAB is also restricted by the accuracy of ZEROPOLY.

Example

In this example, a Butterworth filter is used since it is likely to have some of its zeros located on or just outside the unit circle, and therefore be marginally unstable. The Butterworth filter is converted to a stable filter, one with all its poles and zeros inside the unit circle, using the P_STAB function.

```
h = IIRDESIGN(8, 0.5, /Butter)
    ; Create a Butterworth filter.
PARSEFILT, h, name, num, den
    ; Extract the numerator and denominator polynomials from the filter.
new_num = P_STAB(num)
new_den = P_STAB(den)
    ; Stabilize the numerator and denominator polynomials.
new_den = new_den/new_den(0)
    ; Scale the denominator so that a0 = 1.0.
```

```
new_num = TOTAL(new_den)/TOTAL(new_num)*new_num
```
; Scale the denominator so that h_1 = 1.0, which is the same thing
; as saying that the filter amplitude at zero frequency is unity.

```
newh = FILTSTR(new_num, new_den)
```
; Make a new filter out of the new numerator and denominator
; polynomials.

```
resp = FREQRESP_Z(newh, Outfreq = f)
PLOT, f, ABS(resp)
```
; Plot the frequency response of this new filter just to show that it
; has the same characteristics as the original (*Figure 2-31*).

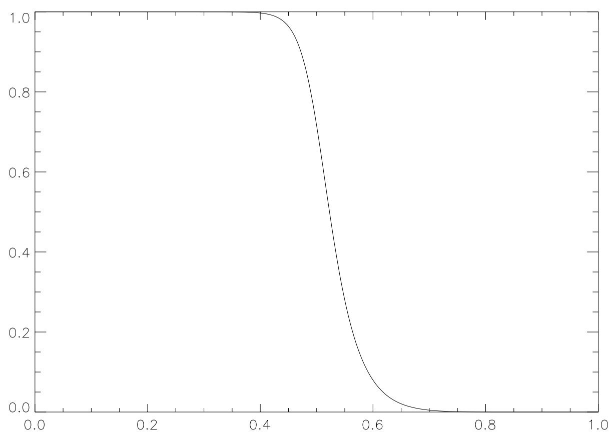


Figure 2-31 Frequency response plot of the stabilized Butterworth filter. Note that the frequency response of the stabilized filter is identical to the original, default Butterworth filter created using IIRDESIGN.

See Also

ZEROPOLY

For Additional Information

Roberts and Mullis, 1987, p. 202.

P_SUM Function

Computes the sum of two polynomials.

Usage

result = P_SUM(*a*, *b*)

Input Parameters

a — The array of coefficients of the first polynomial.

b — The array of coefficients of the second polynomial.

Returned Value

result — An array of coefficients representing the sum of two polynomial coefficient arrays.

Keywords

None.

Discussion

Given two polynomials $A(z)$ and $B(z)$, such as

$$A(z) = a_0 + a_1 z^{-1} + a_2 z^{-2} + \ldots + a_N z^{-N}$$

and

$$B(z) = b_0 + b_1 z^{-1} + b_2 z^{-2} + \ldots + b_M z^{-M} ,$$

their polynomial sum is simply

$$C(z) = A(z) + B(z) .$$

The degree of the resulting polynomial $C(z)$ is the maximum of the degrees (N and M) of the two supplied polynomials $A(z)$ and $B(z)$.

Example

In this example, two polynomials, $A(z) = 1 + 3z^{-1}$,
and $B(z) = 1 + 2z^{-1} + 2z^{-2} + z^{-3}$ are added together using P_SUM.

```
a = [1, 3]
b = [1, 2, 2, 1]
c = P_SUM(a, b)
PM, c
    2
    5
    2
    1
```

QMF Procedure

Applies a perfect reconstruction quadrature mirror filter (QMF) structure to a data sequence.

Usage

QMF, *h*, *x*, *n*, *x0*, *x1*

QMF, *h*, *y*, *n*, *x0*, *x1*, */Backward*

Input Parameters

h — A finite impulse response (FIR) quadrature mirror filter structure.

Parameters for the forward QMF computation (default):

> ***x*** — A data sequence to be processed.
>
> ***n*** — (scalar) The length of the data sequence to process.

Parameters for the backward QMF computation:

> ***x0*** — The lowpass filtered and decimated signal.
>
> ***x1*** — The highpass filtered and decimated signal.
>
> ***n*** — (scalar) The length of the reconstructed data sequence.

Output Parameters

Specific to the forward QMF computation (default):

> ***x0*** — The filtered and decimated lowpass signal.
>
> ***x1*** — The filtered and decimated highpass signal.

Specific to the backward QMF computation:

> ***y*** — The reconstructed data sequence.

Keywords

Backward — If present and nonzero, the backward QMF operation is computed.

Discussion

QMF realizes the filter structures shown in *Figure 2-32*. The definitions of the multirate filtering operations shown in the figure are in the reference pages for FILTUP and FILTDOWN.

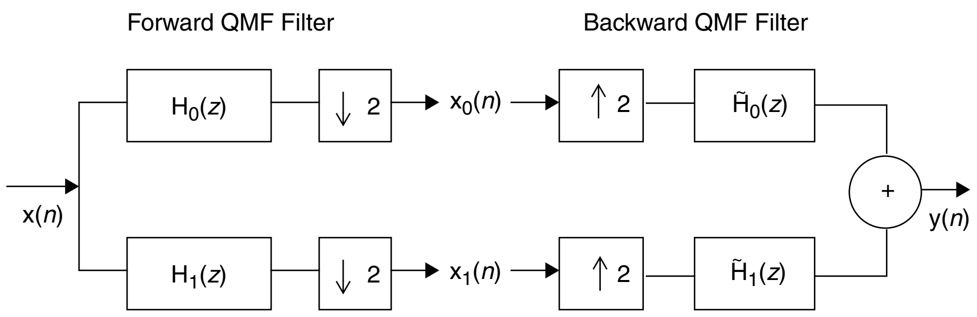


Figure 2-32 Signal flows for the forward and backward QMF procedures.

The four FIR filters shown in *Figure 2-32* are obtained from a single QMF filter that satisfies

$$|H(z)|^2 + |H(-z)|^2 = 1 .$$

Such a QMF may be obtained using QMFDESIGN. The four filters are given by

$$H_0(z) = 2^{1/2}H(z)$$

$$H_1(z) = z^{-N}H_0(-z^{-1})$$

$$\tilde{H}_0(z) = z^{-N}H_0(z^{-1})$$

and

$$\tilde{H}_1(z) = z^{-N}H_1(z^{-1}),$$

where N is the order of $H(z)$. The filter $H_0(z)$ is generally a lowpass filter, and $H_1(z)$ is generally a highpass filter.

The forward QMF computation (default) uses the first half of *Figure 2-32*. A QMF filter $H(z)$ and an input signal x are used to compute the two output signals $x_0(n)$ and $x_1(n)$. The backward QMF computation uses the second half of *Figure 2-32*. The backward, or inverse operation uses a QMF filter $H(z)$ and two input signals $x_0(n)$ and $x_1(n)$ to compute the output signal $y(n)$.

CAUTION QMF does not verify that the filter passed to it is actually a perfect reconstruction QMF. However, if the filter was created using QMFDESIGN and the Caution note on page 181 in the *Discussion* is followed, the filter will be a perfect reconstruction.

Example

This example illustrates how a QMF is designed and then applied to a simple sine wave signal.

```
k = [1, 4, 6, 4, 1]

h = QMFDESIGN(k)

hf = FREQRESP_Z(h, Outfreq = f)

PLOT, f, ABS(hf), Title = 'Quadrature Mirror Filter', $
    XTitle = 'Frequency', YTitle = 'Magnitude'
        ; Plot the designed QMF (Figure 2-33).
```

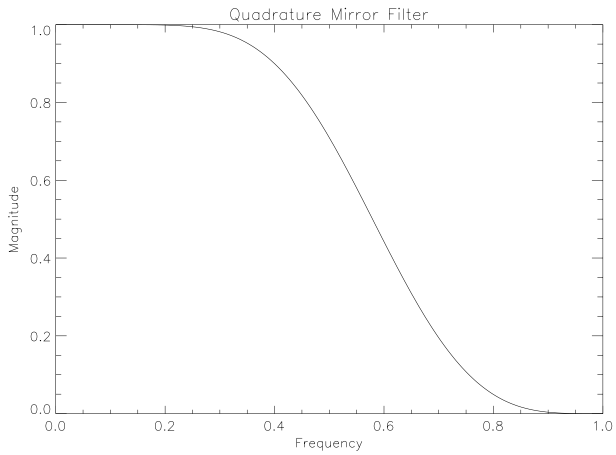
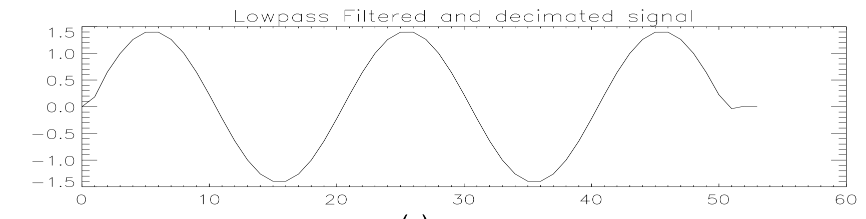


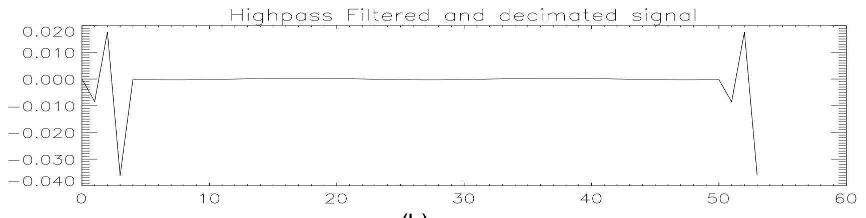
Figure 2-33 Frequency response of the quadrature mirror filter.

```
x = SIN(0.05*!Pi*FINDGEN(100))
```

```
n = N_ELEMENTS(x)
    ; Generate a test signal.
QMF, h, x, n, x0, x1
    ; Forward QMF operation.
!P.Multi = [0, 1, 2]
PLOT, x0, Title =$
    'Lowpass filtered and decimated signal'
PLOT, x1, Title =$
    'Highpass filtered and decimated signal'
        ; Plot the output signals (Figure 2-34).
QMF, h, y, n, x0, x1, /Backward
    ; Backward QMF operation.
PM, MAX(ABS(x-y)), Title = $
    'Maximum Reconstruction Error'
    Maximum Reconstruction Error
        4.4408921e-15
```



(a)



(b)

Figure 2-34 The original signal is forward processed with the QMF (*Figure 2-33*) to produce (a) $x_0(n)$, the lowpass filtered and decimated signal, and (b) $x_1(n)$, the highpass filtered and decimated signal.

See Also

FILTDOWN, FILTUP, QMFDESIGN

For Additional Information

Akansu and Haddad, 1992.

Vaidyanathan, 1993.

QMFDESIGN Function

Designs a finite impulse response (FIR) quadrature mirror filter (QMF).

Usage

result = QMFDESIGN(*k*)

Input Parameters

k — An array containing the coefficients of a known polynomial factor of the desired QMF.

Returned Value

result — A filter structure containing the coefficients of the QMF. If the known polynomial factor $K(z)$ has order M , the QMF filter has an order that is $\leq 2M - 1$.

Keywords

None.

Discussion

A quadrature mirror filter $H(z)$ is defined by the equation

$$|H(z)|^2 + |H(-z)|^2 = 1 .$$

Given a known factor $K(z)$ of $H(z)$, QMFDESIGN finds the factor $U(z)$ so that $H(z) = K(z)U(z)$ satisfies the QMF equation above.

The algorithm used to solve this problem first solves the equation

$$|K(z)|^2 X(z) + |K(-z)|^2 X(-z) = z^M$$

for $X(z)$ using the Euclid algorithm discussed in Demeure and Mullis (1989). Next, a spectral factorization of $X(z)$ is performed to obtain

$X(z) = U(z)U(z^{-1})$ using the algorithm discussed in Demeure and Mullis (1990). Finally, the QMF $H(z)$ is determined as $H(z) = K(z)U(z)$.

By choosing

$$K(z) = (1 + z^{-1})^M$$

QMFDESIGN generates the Daubechies, compactly supported, orthonormal wavelet with M vanishing moments.

Another common choice is to select $K(z)$ so that it has all its zeros on the unit circle in the left half-plane to approximate the ideal half-band lowpass filter.

CAUTION The polynomial $K(z)$ must be chosen so that $K(z)$ and $K(-z)$ are co-prime, and $X(z)$ is positive on the unit circle. If $K(z)$ and $K(-z)$ are not co-prime, a message saying, *Input polynomials are not co-prime* appears. If $X(z)$ is not positive, a message saying, *Newton-Raphson iterations did not converge* appears.

Example

In this example, the Daubechies QMF is designed with 3-zeros and $z = -1$.

```
k = [1, 3, 3, 1]
```

; The coefficients of the polynomial $K(z) = (1+z^{-1})^3$.

```
h = QMFDESIGN(kz)
```

; Design the quadrature mirror filter.

```
hf = FREQRESP_Z(h, Outfreq = f)
```

```
PLOT, f, ABS(hf), $
   Title = 'Daubechies QMF Magnitude', $
   XTitle = 'Frequency'
```

; Plot the filter magnitude response versus frequency
; (*Figure 2-35*).

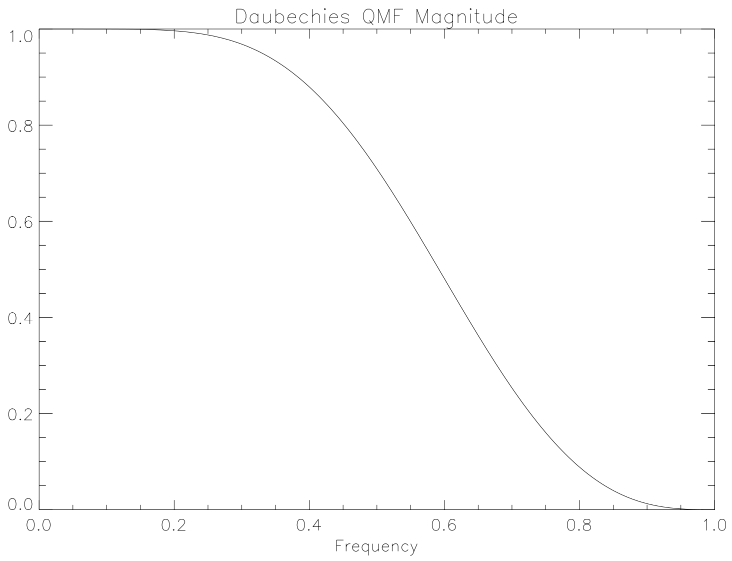


Figure 2-35 Magnitude response of a Daubechies quadrature mirror filter.

See Also

QMF

For Additional Information

Akansu and Haddad, 1992.

Daubechies, 1992.

Demeure and Mullis, 1990.

Duell, 1994.

Vaidyanathan, 1993.

QUADPROG Function

Solves a quadratic programming (QP) problem subject to linear equality or inequality constraints.

Usage

result = QUADPROG(*a*, *b*, *g*, *h*)

Input Parameters

a — A two-dimensional matrix containing the linear constraints.

b — A one-dimensional matrix of the right-hand sides of the linear constraints.

g — A one-dimensional array of the coefficients of the linear term of the objective function.

h — A two-dimensional array of size N_ELEMENTS(*g*)-by-N_ELEMENTS(*g*) containing the Hessian matrix of the objective function. The array must be symmetric positive definite. If *h* is not positive definite, the algorithm attempts to solve the QP problem with *h* replaced by *h* + *diag*1*, such that *h* + *diag*1* is positive definite.

Returned Value

result —The solution to the QP problem.

Keywords

Diag — The name of the variable into which the scalar, equal to the multiple of the identity matrix added to *h* to give a positive definite matrix, is stored.

Double — If present and nonzero, double precision is used in the computation.

Dual — The name of the variable into which an array with N_ELEMENTS(*g*) elements, containing the Lagrange multiplier estimates, is stored.

Meq — The number of linear equality constraints. If *Meq* is used, the equality constraints are located at *A* (*i*, *) for *i* = 0, …, *Meq* – 1.
(Default: *Meq* = N_ELEMENTS(*A* (*, 0)) *n*; i.e., all constraints are equality constraints)

Obj — The name of the variable into which the optimal object function is stored.

Discussion

QUADPROG is based on M.J.D. Powell's implementation of the Goldfarb and Idnani dual quadratic programming (QP) algorithm for convex QP problems subject to general linear equality/inequality constraints (Goldfarb and Idnani, 1983). These problems are of the form

$$\min_{x \in \mathbf{R}^n} g^T x + \frac{1}{2} x^T H x$$

subject to

$$A_1 x = b_1$$

$$A_2 x \geq b_2$$

given the arrays b_0, b_1, and g, and the matrices H, A_0, and A_1. Matrix H is required to be positive definite. In this case, a unique x solves the problem, or the constraints are inconsistent. If H is not positive definite, a positive definite perturbation of H is used in place of H. For more details, see Powell (1983, 1985).

If a perturbation of H, $H + \alpha I$, is used in the QP problem, $H + \alpha I$ also should be used in the definition of the Lagrange multipliers.

Example

The QP problem

$$\min f(x) = x_0^2 + x_1^2 + x_2^2 + x_3^2 + x_4^2 - 2x_1x_2 - 2x_3x_4 - 2x_0$$

subject to

$$x_0 + x_1 + x_2 + x_3 + x_4 = 5$$

$$x_2 - 2x_3 - 2x_4 = -3$$

is solved.

```
RM, a, 2, 5
    row 0: 1 1 1  1  1
    row 1: 0 0 1 -2 -2
        ; Define the coefficient matrix A.
h = [[2, 0, 0, 0, 0], [0,  2, -2,  0,  0], $
    [0, -2, 2, 0, 0], [0, 0, 0, 2, -2], $
    [0, 0, 0, -2, 2]]
        ; Define the Hessian matrix of the objective function. Notice
        ; that since h is symmetric, the array concatenation
        ; operators "[ ]" are used in its definition.
b = [5, -3]
g = [ -2, 0, 0, 0, 0]
x = QUADPROG(a, b, g, h)
PM, x
    1.00000
    1.00000
    1.00000
    1.00000
    1.00000
```

For Additional Information

Goldfarb and Idnani, 1983.

Powell, 1983 and 1985.

RANDOM Function

Generates pseudorandom numbers. The default distribution is a uniform (0, 1) distribution, but many different distributions can be specified through the use of keywords.

Usage

result = RANDOM(*n*)

Input Parameters

n — Number of random numbers to generate.

Returned Value

result — A one-dimensional array of length *n* containing the random numbers. If the keywords *Mvar_Normal* and *Covariances* are used, then a two-dimensional array is returned.

Keywords

A — Shape parameter of the Gamma distribution. Keyword *A* must be positive. Keywords *A* and *Gamma* both must be specified to force RANDOM to return random numbers from a Gamma distribution.

Beta — If present and nonzero, the random numbers are generated from a beta distribution. Keywords *Beta*, *Pin*, and *Qin* all must be specified to force RANDOM to return numbers from a beta distribution.

Covariances — Two-dimensional, square matrix containing the variance-covariance matrix. The two-dimensional array returned by RANDOM is of the following size:

n-by-N_ELEMENTS(*Covariances*(*, 0))

Keywords *Mvar_Normal* and *Covariances* must be specified to return numbers from a multivariate normal distribution.

Double — If present and nonzero, double precision is used.

Exponential — If present and nonzero, the random numbers are generated from a standard exponential distribution.

Gamma — If present and nonzero, the random numbers are generated form a standard Gamma distribution. Keywords *Gamma* and *A* both must be specified to force RANDOM to return random numbers from a Gamma distribution.

Mvar_Normal — If present and nonzero, the random numbers are generated from a multivariate normal distribution. Keywords *Mvar_Normal* and *Covariances* must be specified to return numbers from a multivariate normal distribution.

Normal — If present and nonzero, the random numbers are generated from a standard normal distribution using an inverse CDF method.

Pin — First parameter of the beta distribution. Keyword *Pin* must be positive. Keywords *Pin*, *Qin*, and *Beta* all must be specified to force RANDOM to return numbers from a beta distribution.

Poisson — If present and nonzero, the random numbers are generated from a Poisson distribution. Keywords *Poisson* and *Theta* both must be specified to force RANDOM to return random numbers from a Poisson distribution.

Qin — Second parameter of the beta distribution. Keyword *Qin* must be positive. Keywords *Qin*, *Pin*, and *Beta* all must be specified to force RANDOM to return numbers from a beta distribution.

Theta — Mean of the Poisson distribution. Keyword *Theta* must be positive. Keywords *Theta* and *Poisson* both must be specified to force RANDOM to return random numbers from a Poisson distribution.

Uniform — If present and nonzero, the random numbers are generated from a uniform (0, 1) distribution. The default action of this returns random numbers from a uniform (0, 1) distribution.

Discussion

RANDOM returns random numbers from any of a number of different distributions. The determination of which distribution to generate the random numbers from is based on the presence of a keyword, or group of keywords. If RANDOM is called without any keywords, then random numbers from a uniform (0, 1) distribution are returned.

Uniform (0,1) Distribution

The default action of RANDOM generates pseudorandom numbers from a uniform (0, 1) distribution using a multiplicative, congruent method. The form of the generator follows:

$$x_i \equiv c x_{i-1} \mathrm{mod}(2^{31}-1)$$

Each x_i is then scaled into the unit interval (0, 1). The possible values for c in the generators are 16807, 397204094, and 950706376. The selection is made by using the RANDOMOPT procedure with the *Gen_Option* keyword. The choice of 16807 results in the fastest execution time. If no selection is made explicitly, the functions use the multiplier 16807.

The RANDOMOPT procedure (page 192) called with the *Set* keyword is used to initialize the seed of the random-number generator.

You can select a shuffled version of these generators. In this scheme, a table is filled with the first 128 uniform (0,1) numbers resulting from the simple multiplicative congruent generator. Then, for each x_i from the simple generator, the low-order bits of x_i are used to select a random integer, j, from 1 to 128. The j-th entry in the table is then delivered as the random number, and x_i, after being scaled into the unit interval, is inserted into the j-th position in the table.

The values returned are positive and less than 1.0. Some values returned may be smaller than the smallest relative spacing; however, it may be the case that some value, for example $r(i)$, is such that
$1.0 - r(i) = 1.0$.

Deviates from the distribution with uniform density over the interval (a, b) can be obtained by scaling the output.

Normal Distribution

Calling RANDOM with keyword *Normal* generates pseudorandom numbers from a standard normal (Gaussian) distribution using an inverse CDF technique. In this method, a uniform (0, 1) random deviate is generated. Then, the inverse of the normal distribution function is evaluated at that point using the PV-WAVE function NORMALCDF with keyword *Inverse*.

Deviates from the normal distribution with mean specific mean and standard deviation can be obtained by scaling the output from RANDOM.

Exponential Distribution

Calling RANDOM with keyword *Exponential* generates pseudorandom numbers from a standard exponential distribution. The probability density function is $f(x) = e^{-x}$, for $x > 0$. RANDOM uses an antithetic inverse CDF technique. In other words, a uniform random deviate U is generated, and the inverse of the exponential cumulative distribution function is evaluated at $1.0 - U$ to yield the exponential deviate.

Poisson Distribution

Calling RANDOM with keywords *Poisson* and *Theta* generates pseudorandom numbers from a Poisson distribution with positive mean *Theta*. The probability function (with θ = *Theta*) follows:

$$f(x) = (e^{-\theta}\theta^x)/x!\ , \quad \text{for } x = 0, 1, 2, \ldots$$

If *Theta* is less than 15, RANDOM uses an inverse CDF method; otherwise, the PTPE method of Schmeiser and Kachitvichyanukul (1981) is used. (See also Schmeiser, 1983.) The PTPE method uses a composition of four regions, a triangle, a parallelogram, and two negative exponentials. In each region except the triangle, acceptance/rejection is used. The execution time of the method is essentially insensitive to the mean of the Poisson.

Gamma Distribution

Calling RANDOM with keywords *Gamma* and *A* generates pseudorandom numbers from a Gamma distribution with shape parameter $a = A$ and unit scale parameter. The probability density function follows:

$$f(x) = \frac{1}{\Gamma(a)} x^{a-1} e^{-x} \qquad \text{for } x \geq 0$$

Various computational algorithms are used depending on the value of the shape parameter a. For the special case of $a = 0.5$, squared and halved normal deviates are used; for the special case of $a = 1.0$, exponential deviates are generated. Otherwise, if a is less than 1.0, an acceptance-rejection method due to Ahrens, described in Ahrens and Dieter (1974), is used. If a is greater than 1.0, a 10-region rejection procedure developed by Schmeiser and Lal (1980) is used.

The Erlang distribution is a standard Gamma distribution with the shape parameter having a value equal to a positive integer; hence, RANDOM generates pseudorandom deviates from an Erlang distribution with no modifications required.

Beta Distribution

Calling RANDOM with keywords *Beta*, *Pin*, and *Qin* generates pseudorandom numbers from a beta distribution with parameters *Pin* and *Qin*, both of which must be positive. With $p = Pin$ and $q = Qin$, the probability density function is

$$f(x) = \frac{\Gamma(p+q)}{\Gamma(p)\Gamma(q)} x^{p-1} (1-x)^{q-1}$$

where $\Gamma(\cdot)$ is the Gamma function.

The algorithm used depends on the values of p and q. Except for the trivial cases of $p = 1$ or $q = 1$, in which the inverse CDF method is used, all the methods use acceptance/rejection. If p and q are both less than 1, the method of Jöhnk (1964) is used. If either p or q is less than 1 and the other is greater than 1, the method of Atkinson (1979) is used. If both p and q are greater than 1, algorithm BB of Cheng (1978), which requires very little setup time, is used if x is less than 4, and algorithm B4PE of Schmeiser and Babu (1980) is used if x is greater than or equal to 4.

NOTE Note that for p and q both greater than 1, calling RANDOM to generate random numbers from a beta distribution a loop getting less than four variates on each call yields the same set of deviates as executing one call and getting all the deviates at once.

The values returned are less than 1.0 and greater than ε, where ε is the smallest positive number such that $1.0 - \varepsilon < 1.0$.

Multivariate Normal Distribution

Calling RANDOM with keywords *Mvar_Normal* and *Covariances* generates pseudorandom numbers from a multivariate normal distribution with mean array consisting of all zeros and variance-covariance matrix defined using keyword *Covariances*. First, the Cholesky factor of the variance-covariance matrix is computed. Then, independent random normal deviates with mean zero and variance 1 are generated, and the matrix containing these deviates is post-multiplied by the Cholesky factor. Because the Cholesky factorization is performed in each invocation, it is best to generate as many random arrays as needed at once.

Deviates from a multivariate normal distribution with means other than zero can be generated by using RANDOM with keywords *Mvar_Normal* and *Covariances*, then adding the arrays of means to each row of the result.

Example

In this example, RANDOM is used to generate five pseudorandom, uniform numbers. Since RANDOMOPT is not called, the generator used is a simple multiplicative congruential one with a multiplier of 16807.

```
RANDOMOPT, Set = 123457
    ; Set the random seed.
r = RANDOM(5)
    ; Compute the random numbers.
PM, r
    0.966220
    0.260711
    0.766262
    0.569337
    0.844829
```

See Also

RANDOMOPT

In the *PV-WAVE Reference*:

NORMALCDF

For Additional Information

Ahrens and Dieter, 1974.

Atkinson, 1979.

Cheng, 1978.

Schmeiser, 1983.

Schmeiser and Babu, 1980.

Schmeiser and Lal, 1980.

Schmeiser and Kachitvichyanukul, 1981.

RANDOMOPT Procedure

Sets or retrieves the random number seed to select the uniform (0, 1) multiplicative, congruential pseudorandom-number generator.

Usage

RANDOMOPT

Input Parameters

Procedure RANDOMOPT does not have any positional input parameters. Keywords are required for specific actions to be taken.

Keywords

Gen_Option — An indicator of the generator. The random-number generator is a multiplicative, congruential generator with modulus $2^{31} - 1$. Keyword *Gen_Option* is used to choose the multiplier and to determine whether or not shuffling is done.

Random Number Generator Options

Gen_Option	**Generator**
1	multiplier 16807 used (default)

Random Number Generator Options

Gen_Option	**Generator**
2	multiplier 16807 used with shuffling
3	multiplier 397204094 used
4	multiplier 397204094 used with shuffling
5	multiplier 950706376 used
6	multiplier 950706376 used with shuffling

Get — A named variable into which the value of the current random-number seed is stored.

Set — The seed of the random-number generator. The seed must be in the range (0, 2147483646). If the seed is zero, a value is computed using the system clock; hence, the results of programs using the random-number generators are different at various times.

Discussion

RANDOMOPT is designed to allow you to set certain key elements of the random-number generator function RANDOM.

The uniform pseudorandom-number generators use a multiplicative congruential method, with or without shuffling. The value of the multiplier and whether or not to use shuffling are determined by keyword *Gen_Option*. The description of function RANDOM may provide some guidance in the choice of the form of the generator. If no selection is made explicitly, the generators use the multiplier 16807 without shuffling. This form of the generator has been in use for some time (Lewis et al. 1969).

Keyword *Set* is used to initialize the seed used in the random-number generators. The form of the generators follows:

$$x_i = cx_{i-1} \bmod (2^{31} - 1) \ .$$

The value of x_0 is the seed. If the seed is not initialized prior to invocation of any of the routines for random-number generation by calling RANDOMOPT, the seed is initialized via the system clock. The seed can be re-initialized to a clock-dependent value by calling RANDOMOPT with *Set* set to zero.

A common use of keyword *Set* is in conjunction with the keyword *Get* to restart a simulation. Keyword *Get* retrieves the current value of the “seed” used in the random-number generators.

Example

This example illustrates the statements required to restart a simulation using the keywords *Get* and *Set*. The example shows that restarting the sequence of random numbers at the value of the last seed generated is the same as generating the random numbers all at once.

```
seed = 123457

nrandom = 5

RANDOMOPT, Set = seed
```

; Set the seed using the keyword *Set*.

```
r1 = RANDOM(nrandom)

PM, r1, Title = 'First Group of Random Numbers'
   First Group of Random Numbers
   0.966220
   0.260711
   0.766262
   0.569337
   0.844829

RANDOMOPT, Get = seed
```

; Get the current value of the seed using the keyword *Get*.

```
RANDOMOPT, Set = seed

r2 = RANDOM(nrandom)

PM, r2, Title = 'Second Group of Random Numbers'
   Second Group of Random Numbers
   0.0442665
   0.987184
   0.601350
   0.896375
   0.380854

RANDOMOPT, Set = 123457
```

; Reset the seed to the original seed.

```
r3 = RANDOM(2 * nrandom)

PM, r3, Title = 'Both Groups of Random Numbers'
   Both Groups of Random Numbers
      0.966220
      0.260711
      0.766262
      0.569337
      0.844829
      0.0442665
```

```
        0.987184
        0.601350
        0.896375
        0.380854
```

See Also

RANDOM

For Additional Information

Lewis, Goodman and Miller, 1969

REFLINES Procedure

Produces reference lines on a frequency response graph.

Usage

REFLINES, *x*

Input Parameters

x — A one-dimensional array of the frequencies on the *x*-axis where reference lines are to be drawn.

Keywords

None.

Discussion

REFLINES is particularly useful for displaying cutoff frequencies on an existing frequency response plot. For each value in the input array, one vertical line is drawn from the *x*-axis to the maximum *y* value. The lines are drawn in color 55, which is dependent on the currently loaded color table.

Example

In this example, a single bandpass filter is plotted, and the band edges are drawn using REFLINES.

```
h = FIRDESIGN(FIRWIN(101, /Blackman), 0.4, 0.6, $
    /Bandpass)
```

; Approximate an ideal bandpass filter with normalized band
; edges at 0.4 and 0.6 using a Blackman window.

```
hf = FREQRESP_Z(h, Outfreq = f)
```

```
PLOT, f, ABS(hf)
```

; Plot the magnitude of the frequency response (*Figure 2-36*).

```
LOADCT, 12
```

; Load the 16 level color table so the reference lines appear green.

```
REFLINES, [0.4, 0.6]
```

; Draw reference lines at the band edge frequencies
; (see *Figure 2-36*).

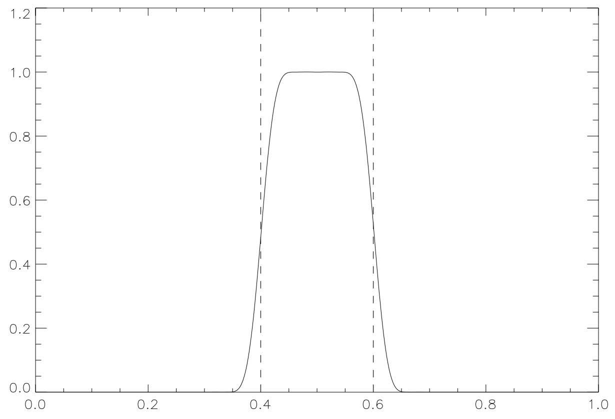


Figure 2-36 Use of reference lines to indicate the edge frequencies of a bandpass filter.

REMEZ Function

Designs an optimal linear phase FIR digital filter using the Remez exchange algorithm.

Usage

result = REMEZ(*nfilt*, *ftype*, *edge*, *fx*, *wtx* [, *lgrid*][, *iters*])

Input Parameters

nfilt — (scalar) The filter length.

ftype — (scalar) The type of filter. Set *ftype* to 1 for multiple passband/stopband, 2 for differentiator, 3 for Hilbert transformer.

edge — A one-dimensional floating point array specifying the upper and lower cut-off frequencies, up to a maximum of ten bands.

fx — A one-dimensional floating point array of length N_ELEMENTS(*edge*)/2 containing the desired frequency response in each band.

wtx — A one-dimensional floating point array of length N_ELEMENTS(*edge*)/2 containing the positive weight function for each band.

lgrid — (optional) A scalar value to set grid density. (Default: *lgrid* = 16)

iters — (optional) A scalar number between 1 and 25 indicating the maximum number of iterations within the REMEZ exchange algorithm. (Default: *iters* = 25)

Output Parameters

iters — (optional) The number of iterations within the REMEZ exchange algorithm. On output, *iters* is a scalar value containing the actual number of iterations used.

Returned Value

result — A filter structure containing the impulse response of the optimal filter.

Keywords

None.

Discussion

Given a desired magnitude response

$F(e^{j\theta})$,

this function uses Parks' and McClellan's variation of the Remez exchange algorithm to find a linear phase FIR filter

$H(e^{j\theta})$, such that

$$(e^{j\theta}) = \sum_{k=0}^{K-1} h_k e^{-jk\theta} \qquad \theta = \pi f$$

which approximates the desired magnitude response.

The approximation criteria is to minimize the weighted Chebyshev error given by

$$\underset{\theta}{ax}\, W(e^{j\theta})\left|F(e^{j\theta}) - H(e^{j\theta})\right|,$$

where $W(e^{j\theta})$ is a positive weight function.

The specific implementation of this function is based on the algorithm in McClellan, Parks, and Rabiner, 1979. This numerical implementation utilizes samples of the functions $F(e^{j\theta})$ and $W(e^{j\theta})$ given by

$$f_x = [F(e^{j\pi f_0}), F(e^{j\pi f_2}), F(e^{j\pi f_4}), \ldots, F(e^{j\pi f_N})] \text{ and}$$

$$w_x = [W(e^{j\pi f_0}), W(e^{j\pi f_2}), W(e^{j\pi f_4}), \ldots, W(e^{j\pi f_N})]$$

as the input arguments for the frequencies specified by

edge = [f_0, f_1, ..., f_{2N}]

The frequency points must be between 0 and 1, where 1 is the Nyquist frequency. The frequencies must have increasing order.

The value of $(e^{j\pi f_k})$ and $(e^{j\pi f_k})$

defines the desired magnitude response and weight function for the interval (f_k, f_{k+1}) for k = 0, 2, 4, ... The values of the magnitude response and weight function for the intervals (f_k, f_{k+1}) for k = 1, 3, 5, ... are undefined and are so-called “don’t care” regions.

If *ftype* = 1, REMEZ designs Type 1 and Type 2 linear phase FIR filters when *nfilt* is odd or even, respectively. (See FIRLS for the four types of linear phase filters.)

If *ftype* = 3, REMEZ designs Type 3 and Type 4 linear phase FIR filters when *nfilt* is odd or even, respectively.

If *ftype* = 2, REMEZ also designs Type 3 and Type 4 linear phase FIR filters when *nfilt* is odd or even, respectively, but the values of *fx* and *wx* are treated differently. The values of *fx*(k) specify the slope of the desired frequency response

$$(e^{j\pi f_k})$$

on the interval (f_k, f_{k+1}) and the values of *wx*(k) specify the weight function evaluated as

$$\frac{1}{(e^{j\pi f_k})}.$$

In the solution, the values of $(e^{j\pi f_k})$ and $(e^{j\pi f_k})$

are interpolated onto a dense grid of points proportional to *lgrid* * *nfilt*/2. The default value of *lgrid* is 16.

The linear phase filter coefficients are returned in a filter structure

$$H(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2} + \ldots + b_m z^{-m}}{1}$$

where $m = nfilt - 1$.

Example

In this example, REMEZ is used to compute a multiband filter.

```
nfilt = 55

ftype = 1

edge = [.00, .10, .20, .30, .36, .50, .60, .72, $
   .82, 1.00]

fx = [0, 1., 0, 1., 0]

wtx = [10.0, 1., 3., 1., 20.]

h = REMEZ(nfilt, ftype, edge, fx, wtx)

PLOT, FINDGEN(512)/511, ALOG(ABS(FREQRESP_Z(h))), $
   XStyle = 1, Title = $
   'Log magnitude response of multiband filter'
      ; Plot the filter response (Figure 2-37).

OPLOT, [0, 1], [0, 0], COLOR = .5*!D.N_Colors

OPLOT, FINDGEN(512)/511,ALOG(ABS(FREQRESP_Z(h)))
```

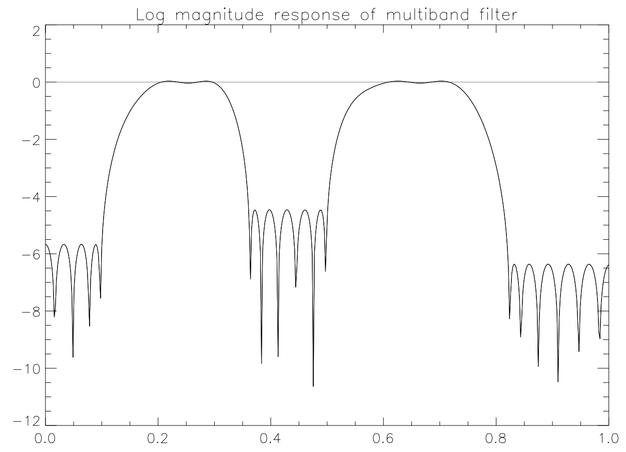


Figure 2-37 Log magnitude response of a multiband filter designed using REMEZ.

For Additional Information

McClellan, Parks, Rabiner, 1979

Oppenheim and Schafer, 1989, section 7.6.

ROOT2POLY Function

Forms a polynomial from a given set of roots.

Usage

result = ROOT2POLY(*r*)

Input Parameters

r — A one-dimensional array or scalar containing the polynomial roots.

Returned Value

result — A one-dimensional array containing the coefficients of the polynomial.

Keywords

None.

Discussion

Given a set of roots r_0, r_1, r_2, ..., r_N, ROOT2POLY determines the coefficients of the polynomial $A(x)$ where $A(x)$ is given by

$$A(x) = \prod_{i=0}^{N} (x - r_i)$$

$$= a_0 + a_1 x + a_2 x^2 + \ldots + a_N x^N \quad , a_N = 1 .$$

ROOT2POLY always returns a polynomial with a_N normalized to 1.

Example

In this example, ZEROPOLY is used to first compute the roots of the polynomi-al$A(x)$ given by

$$A(x) = 3 + 2x + 12x^2 + 4x^3 + x^4 .$$

ROOT2POLY is then used to reproduce $A(x)$ from its roots.

```
A = [3, 2, 12, 4, 1]

r = ZEROPOLY(A)

PM, ROOT2POLY(r), Format = '(f10.1)'
     3.0
     2.0
    12.0
     4.0
     1.0
```

See Also

ZEROPOLY

SCHURCOHN Function

Determines if a filter polynomial is stable using the Schur-Cohn stability test.

Usage

result = SCHURCOHN(*a* [, *epsilon*])

Input Parameters

a — The array of coefficients of the filter polynomial.

epsilon — (optional) A value used to numerically determine if the reflection coefficients equal one. (Default: [*machine epsilon*]$^{1/2}$. It may be set to zero.)

Returned Value

result — A scalar that equals zero if the polynomial is unstable, or one if the polynomial is stable.

Keywords

None.

Discussion

SCHURCOHN determines if the roots of a filter polynomial:

$$A(z) = a_0 + a_1 z^{-1} + a_2 z^{-2} + \ldots + a_N z^{-N}$$

have magnitude less than one. This is accomplished by performing the Schur-Cohn stability test on the polynomial. The test first computes the reflection coefficients of the polynomial. If the magnitude of the reflection coefficients are all less than one then the polynomial is stable.

In this numerical implementation of the Schur-Cohn stability test, the magnitude of the reflection coefficients are tested to see if they are less than 1 – *epsilon*. As an option, you may change the value of *epsilon*. (Default: *epsilon* = (machine epsilon)$^{1/2}$).

Example

In this example, two filter polynomials (one stable and the other unstable) are generated, and then SCHURCOHN is used to test their stability.

```
a1 = P_MULT([1, 0.5], [1, 0.3])
```

; The stable polynomial $a_1 = (1 + 0.5\, z^{-1})\, (1 + 0.3\, z^{-1})$.

```
PM, SCHURCOHN(a1), Title = 'Result of '+ $
   'SCHURCOHN for Stable Filter Polynomial'
      Result of SCHURCOHN for Stable Filter Polynomial
         1
```

```
a2 = P_MULT([1, 0.5], [1, 1.0/0.3])
```

; The unstable polynomial $a_2 = (1 + 0.5\, z^{-1})\, (1 + (1.0/0.3)\, z^{-1})$.

```
PM, SCHURCOHN(a2), Title = $
   'Result of SCHURCOHN for Unstable Filter Polynomial'
      Result of SCHURCOHN for Unstable Filter Polynomial
         0
```

For Additional Information

Proakis and Manolakis, 1992, pp. 288 - 289.

Roberts and Mullis, 1987, p. 527.

SGFDESIGN Function

Designs a finite impulse response (FIR) Savitsky-Golay filter.

Usage

result = SGFDESIGN(*m*, *l*)

Input Parameters

m — The length of the filter.

l — The number of moments preserved by the filter.

Returned Value

result — A filter structure containing the coefficients of the FIR filter.

Keywords

None.

Discussion

SGFDESIGN designs optimal linear phase FIR filters for estimating the signal $s(n)$ from the noise-corrupted observation

$$x(n) = s(n) + e(n) ,$$

where the noise $e(n)$ is an independent, identically distributed random variable.

The coefficients of the FIR filter

$$H(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2} + \ldots + b_m z^{-m}}{1}$$

are chosen so that the signal estimate

$$y(n) = b_n * x(n) = \sum_k x(k) b_{k-n}$$

satisfies two properties. First, the filter minimizes the error

$$E[b_n * e(n)]^2$$

where the operation E denotes the mathematical expectation. Second, the moments of the signal $s(n)$ are conserved up to a desired order L.

The moment constraints can be equivalently stated as requiring the filter coefficients b_n to satisfy

$$\sum_n b_n = 1$$

and

$$\sum_n n^l b_n = 0, \quad l = 1, 2, ..., L.$$

This filter is most often used in the physical sciences for smoothing experimental data.

Example

This example illustrates the design of a Savitzky-Golay filter.

```
h = SGFDESIGN(15, 4)
    ; Design the filter.
hf = FREQRESP_Z(h, Outfreq = f)
PLOT, f, ABS(hf), Title =$
    'Savitzky-Golay Filter Magnitude', $
    XTitle = 'Frequency'
    ; Plot the magnitude of the frequency response (Figure 2-38).
```

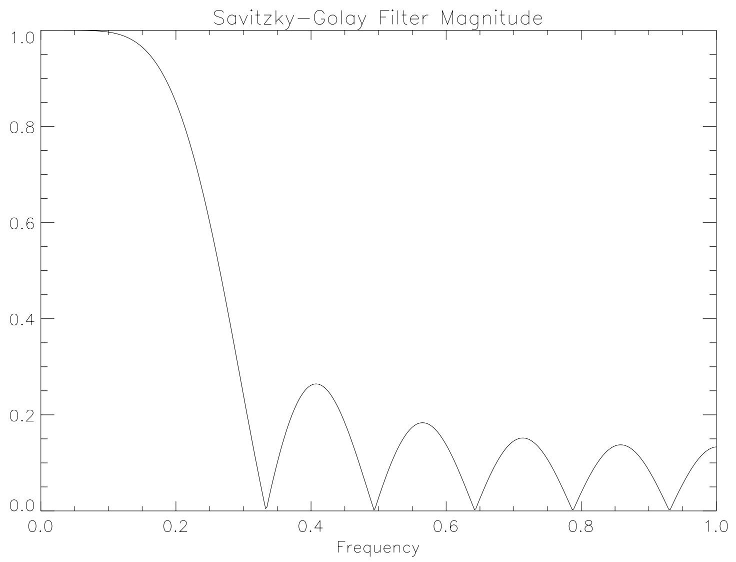


Figure 2-38 Magnitude response of a Savitzky-Golay filter.

For Additional Information

Schussler and Steffen, 1988, Section 8.3.2, pp. 441-443.

Savitsky and Golay, 1964.

Steiner, Termonia and Deltour, 1972.

SIGNAL Function

Computes a variety of standard signals, including a square wave, sine wave, cosine wave, sawtooth wave, random waves, and the periodic sinc or Dirichlet function.

Usage

result = SIGNAL(*n, keyword*)

Input Parameters

n — The number of values to compute. The parameter *n* must be greater than 2.

keyword — Exactly one keyword must be set to select the wave type.

Returned Value

result — A one-dimensional array containing the requested signal.

Keywords

Cosine — If present and nonzero, generates a wave using the cosine function.

Diric — If present and nonzero, generates a wave using the periodic sinc or Dirichlet function.

Norm_random — If present and nonzero, generates a wave using a random normal distribution.

Periods — A scalar value specifying the number of periods to compute for the desired signal.

Sawtooth — If present and nonzero, generates a wave using the sawtooth function with period 2π. If *Sawtooth* is set to a value in the closed interval [0, 1], the peak of the wave occurs at (*Sawtooth**2*!Pi). For example,

> *Sawtooth* = 0.0, results in a pure sawtooth waveform (pulse up, ramp down).
>
> *Sawtooth* = 0.5, results in a triangle waveform (ramp up, ramp down).

Sawtooth = 1.0, results in a pure ramp waveform (ramp up, pulse down).

Sine — If present and nonzero, generates a wave using the sine function.

Square — If present, generates a wave using the square function. The value of *Square* specifies the percent of the wave that is positive. Thus, setting *Square* = 0.5 results in a one-dimensional array in which (*n* +1)/2 elements are equal to 1, and *n*/2 elements are equal to –1.

Unif_random — If present and nonzero, generates a wave using a random uniform distribution.

Discussion

The standard definition of each keyword signal is used to compute the period of the desired signal. Specifically, for a given value of *n*, compute *t*, such that *t* = DINDGEN(*n*)/*n*.

The following definitions are used.

- Sine wave — SIN(*Periods**2π**t*)
- Cosine wave — COS(*Periods**2π**t*)
- Random Uniform wave — RANDOM(*n*, /*Uniform*)
- Random Normal wave — RANDOM(*n*, /*Normal*)
- Square wave — A square wave with a maximum value of 1, and a minimum of –1. The value of the keyword *Square* defines the percent of the wave equal to 1.
- Sawtooth wave — A sawtooth wave with an adjustable peak location based on the value of *Sawtooth*.
- Periodic sinc, or Dirichlet wave —
  If *t* = 2*!Pi**k*, then *Diric* = $(–1)^{(k*n–1)}$,
  otherwise *Diric* = SIN(*n***x*/2) / (*n**SIN(*x*/2).

Example

In this example, SIGNAL is used to generate each of the seven types of signal plots.

```
!P.Multi = [0, 2, 4]

!P.Charsize = 2

n = 500

p = 3

PLOT, SIGNAL(n, /Sine, Periods = p), Title = 'Sine'
```

```
PLOT, SIGNAL(n, /Cosine, Periods = p), Title = $
   'Cosine'

PLOT, SIGNAL(n, Square = .75, Periods = p), $
   Title = 'Square', YRange = [-1.5, 1.5]

PLOT, SIGNAL(n, Sawtooth = .25, Periods = p), $
   Title = 'Sawtooth'

PLOT, SIGNAL(n, /Unif_random), Title = $
   'Random Uniform'

PLOT, SIGNAL(n, /Norm_random), Title = $
   'Random Normal'

PLOT, SIGNAL(n, Diric = 24, Periods = p), Title =$
   'Periodic sinc, or Dirichlet'
      ; The seven plots are shown in Figure 2-39. Note that the periodic
      ; signal plot (Sine, Cosine, Square, Sawtooth and Diric) each show
      ; three periods of the signal by using the keyword Periods in the
      ; calling sequence.
```

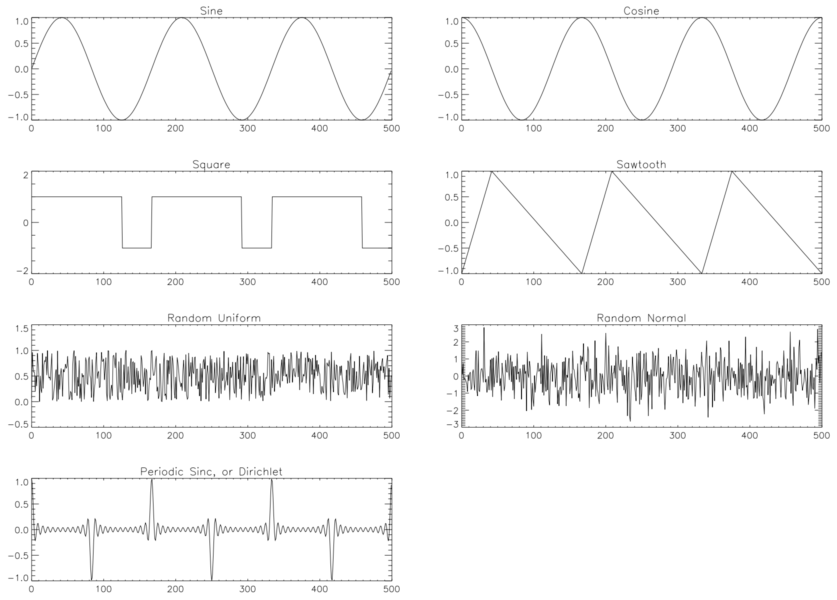


Figure 2-39 Sample signals produced using SIGNAL.

See Also

SINC

SINC Function

Computes the sine of the input divided by the input.

Usage

result = SINC(*x*)

Input Parameters

x — The value at which $\sin(x)/x$ is evaluated.

Returned Value

result — The sine of the input divided by the input.

Keywords

None.

Discussion

SINC computes sin(*x*)/*x* for all $x \in \mathbb{R}$. Specifically,

$$SINC(x) = \begin{cases} \sin(x)/x & |x| \geq \varepsilon \\ 1 & |x| < \varepsilon \end{cases},$$

where ε = (machine precision)$^{1/2}$.

The SINC function is used in the sampling theorem in the frequency domain of the Fourier integral.

Example

In this example SINC is computed for *x* = 0, 1, and π.

```
x = [0.0, 1.0, !Pi]
PM, SINC(x)
     1.0000000
     0.84147098
    -2.7827534e-08
```

See Also

SIGNAL

SPECTROGRAM Function

Computes the spectrogram of a data sequence.

Usage

result = SPECTROGRAM(*x* [, *length*][, *overlap*])

Input Parameters

x — A one-dimensional array containing the data sequence from which the spectrogram will be computed.

length — (optional) The length of the subsections of *x* on which to operate.

overlap — (optional) The number of elements overlapped by successive subsections of *x*. (Default: 0.75**length*)

Returned Value

result — A two-dimensional array containing the spectrogram of *x*.

Keywords

Padding — If present and nonzero and greater than *length*, subsections of *x* are padded with the specified number of zeros.

Print_info — If present and nonzero, the number and length of subsections of *x* are output.

Window_param — The window parameter for a Kaiser window.

Window_type — A scalar value indicating the type of window to use when computing the spectrum of subsections of *x*. See page 71 (the FIRWIN discussion) for a list of available values and window types.

Discussion

The spectrogram is a useful tool for displaying the time-dependent frequency content of a signal. It is also referred to as a Gabor transform, short-time Fourier transform, or waterfall plot.

The spectrogram is determined by breaking the data sequence into length *N* overlapping data segments represented as

$$x(n) = x(n + iR), \qquad \text{for } n = 0, 1, ..., N-1$$

$$i = 0, 1, ..., I-1$$

where $R = (N - overlap)$.

A power spectrum of each data segment is then determined using SPECTRUM. The power spectrum for each data segment is stored as a matrix, with the first index representing the data segment and the second index representing uniform samples of the power spectrum.

The spectrogram is typically displayed as an image with the magnitude of the power spectrum represented by the grey (or color) scale.

Example

In this example, SPECTROGRAM is used on a file containing a signal of a human voice (*Figure 2-40*).

(UNIX) To open the file on a UNIX system:

```
OPENR, u, GETENV('VNI_DIR')+ $
   '/sigpro-1_1/test/voice.dat', /Get_Lun
```

(OpenVMS) To open the file on an OpenVMS system:

```
OPENR, u, GETENV('VNI_DIR')+ $
   '[SIGPRO-1_1.TEST]VOICE.DAT', /Get_Lun
```

(Windows) To open the file on a Windows system:

```
OPENR, u, GETENV('VNI_DIR')+ $
   '\sigpro-1_1\test\voice.dat', /Get_Lun
x = BYTARR(7519)
READU, u, x
CLOSE, u
xs = 400
ys = 400
WINDOW, XSize = xs, YSize = ys
PLOT, x, Position = [0, .5, 1, 1], $
   XStyle = 5, YStyle = 5, /Normal
      ; Plot the original voice signal (Figure 2-40 (a)).
mat = SPECTROGRAM(x, 256)
   ; Compute spectrogram of the original signal.
TVSCL, Hist_Equal(CONGRID(ALOG10(mat > 1.e-5),$
   xs, ys/2))
```

```
; Display the spectrogram as an image (Figure 2-40 (b)).
```

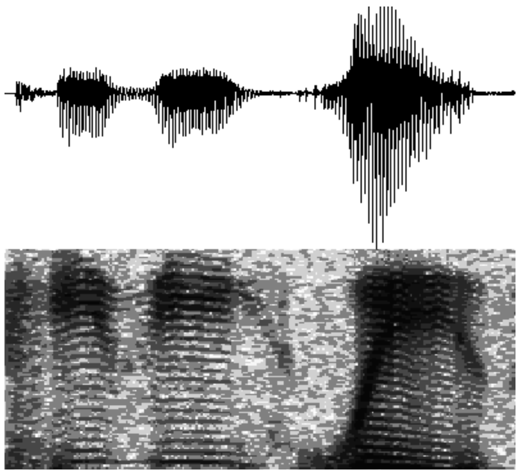


Figure 2-40 A voice signal (a) is processed with SPECTROGRAM and displayed in PV‑WAVE as an image (b).

See Also

SPECTRUM, WAVELET

For Additional Information

Nawab and Quatieri, 1988.

Oppenheim and Schafer, 1975.

SPECTRUM Function

Estimates the power spectrum (power spectral density) of a data sequence.

Usage

result = SPECTRUM(*x*, *length* [, *overlap*])

Input Parameters

x — A one-dimensional array containing the sequence from which the power spectrum will be computed.

length — The length of the subsections of *x* on which to operate. (Default: *N_ELEMENTS*(*x*)/5)

overlap — (optional) The number of elements the successive subsections of *x* will overlap. (Default: 0.5**length*)

Returned Value

result — A one-dimensional array containing the power spectrum of *x*.

Keywords

Padding — If present and nonzero and greater than *length*, the subsections of *x* are padded with the specified number of zeros.

Print_info — If present and nonzero, returns the number and length of subsections of *x*.

Window_param — Window parameter for a Kaiser window.

Window_type — A scalar value indicating the type of window to use when computing the spectrum of subsections of *x*. See page 71 (the FIRWIN discussion) for the list of values and window types available.

Discussion

SPECTRUM computes one of several different power spectrum estimates *P*(*f*) depending on the input parameters. Specific estimates available include the periodogram, the modified periodogram, Bartlett's method, and Welch's method. In all cases uniform samples of the power spectrum are returned. If the parameter *length* = L, the frequency sample values are

$$f_k = k/L\ , \quad k = 0, 1, ..., M \quad \text{for real data.}$$

where $M = [(L + 2)/2]$ for L even and $M = [(L + 1)/2]$ for L odd for real data.

For complex data, $M = L$.

The periodogram is defined as

$${}_{p}(f) = \frac{1}{L}\left|\sum_{l=0}^{L-1} x(l)e^{-j\pi fl}\right|^2,$$

where the frequency variable f is normalized to the Nyquist frequency of 1.0. To obtain uniform samples of the periodogram using SPECTRUM, set *length* equal to the length of the data x, and set *Window_type* to rectangular.

The modified periodogram is defined as

$${}_{mp}(f) = \frac{1}{L}\left|\sum_{l=0}^{L-1} w(l)x(l)e^{-j\pi fl}\right|^2,$$

where $w(l)$ is a data window sequence. To obtain uniform samples of the modified periodogram using SPECTRUM, set *length* equal to the length of the data x and set *Window_type* to any of the windows discussed in FIRWIN (page 70).

Bartlett's method breaks the data into non-overlapping data segments represented as

$$x_i(n) = x(n + iL) \quad n = 0, 1, \ldots , L - 1 \quad i = 0, 1, \ldots , I - 1.$$

A periodogram

$$P_i(f) = \left|\sum_{n=0}^{L-1} x_i(n)e^{-j\pi fn}\right|^2$$

is computed for each data segment and averaged to obtain the Bartlett estimate

$$P_B(f) = \frac{1}{I}\sum_{i=1}^{I-1} P_i(f).$$

To obtain uniform samples of Bartlett's estimate using SPECTRUM, set *length* to be less than the data length, set *overlap* to 0 and set *Window_type* to rectangular.

The Welch method breaks the data into length L overlapping data segments represented as $(n + iL)$.

$$x_i(n) = x(n + iQ) \qquad n = 0, 1, \ldots, L-1 \qquad i = 0, 1, \ldots, I-1$$

where $Q = (L - overlap)$.

A modified periodogram is then computed for each data segment given by

$$P_i(f) = \frac{1}{Lu}\left|\sum_{n=0}^{L-1} x_i(n)w(n)e^{-j\pi fn}\right|^2$$

where

$$u = \frac{1}{L}\sum_{n=0}^{L-1} w^2(n) .$$

The Welch power spectrum estimate is the average of the modified periodogram of each data segment, given by

$$P_w(f) = \frac{1}{I}\sum_{i=1}^{I-1} P_i(f) .$$

To obtain uniform samples of the Welch estimate using SPECTRUM, set *length* less than the data length, set *overlap* to a nonzero value and set *Window_type* to any of the windows discussed in FIRWIN (page 70).

NOTE In estimating the power spectrum it is assumed that the input signal is stationary (i.e., the frequency content does not change with time). If the signal is nonstationary, the functions SPECTROGRAM and WAVELET can often provide better results than SPECTRUM.

Example

In this example, a multiple bandpass filter is designed and the power spectral density is estimated using SPECTRUM.

```
!P.Multi = [0, 1, 2]
```

```
f = [0, .18, .2, .22, .38, .4, .42, .58, .6, .62, $
    .78, .8, .82, 1]

ampl = [0, 0, 5, 10, 10, 5, 0, 0, .5, 1, 1, .5,$
    0, 0]

h = FIRLS(101, f, ampl, /Freqsample)

hf = FREQRESP_Z(h, Outfreq = f)

PLOT, f, 10.0d*ALOG(ABS(hf)^2), Title =$
    'Filter Response', XTitle = 'Frequency',$
    YTitle = 'Magnitude (dB)'
        ; Plot a multiple bandpass filter (Figure 2-41 (a)).

n = 10000

Length = 1024

Overlap = Length/2

RANDOMOPT, Set = 31

x = RANDOM(n, /Normal)
    ; Generate white noise.

y = FILTER(h, x)
    ; Generate colored noise.

sy = SPECTRUM(y, Length, Overlap)

m = FLOAT(N_ELEMENTS(sy))

PLOT, FINDGEN(m)/m, 10.0d*ALOG(sy), $
    Title = 'Power Spectrum Estimate', $
    XTitle = 'Frequency',$
    YTitle = 'Magnitude (dB)'
        ; Plot the power spectral density of the multiple bandpass filter
        ; (see Figure 2-41 (b)).
```

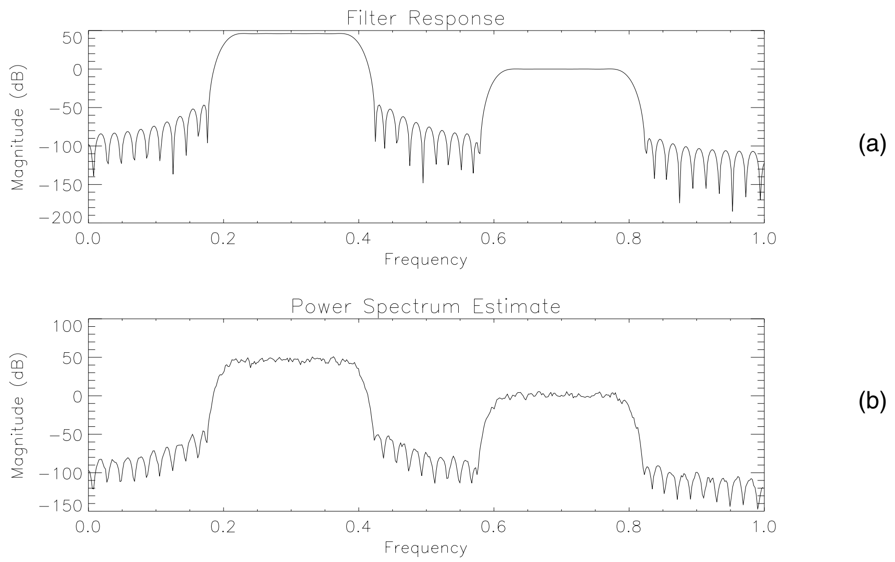


Figure 2-41 (a) A plot of a multiple bandpass filter, and (b) the plot of its power spectral density estimate.

See Also

SPECTROGRAM, WAVELET

For Additional Information

Oppenheim and Schafer, 1975.

TOEPSOL Function

Solves symmetric Toeplitz linear equations using Levinson's algorithm.

Usage

result = TOEPSOL(*r*, *b*)

Input Parameters

r — The first row of the Toeplitz matrix.

b — The right-hand side of a linear equation.

Returned Value

result — An array containing the solution of the linear equation.

Keywords

None.

Discussion

Given the first row of the Toeplitz matrix

$$R(m) = \begin{bmatrix} r(0) & r(1) & & r(m) \\ r(1) & \ddots & \ddots & \\ & \ddots & \ddots & r(1) \\ r(m) & & r(1) & r(0) \end{bmatrix}$$

and the right hand side array $b = [b(0), b(1), \ldots, b(m)]^T$, TOEPSOL is a fast algorithm for solving the linear equation

$$Rx = b\ ,$$

for the array x.

NOTE TOEPSOL is for symmetric, positive, definite matrices only.

Example

TOEPSOL is one of a suite of functions (JURYRC, LEVCORR, LEVDURB, and TOEPSOL) used to solve Toeplitz linear equations and factorization problems. For an example demonstrating the use of TOEPSOL, see the description of JURYRC (page 129).

For Additional Information

Proakis and Manolakis, 1992.

Roberts and Mullis 1987.

WAVELET Function

Computes the wavelet transform of a data sequence using compactly supported orthonormal wavelets.

Usage

result = WAVELET(*h*, *x*, *n*)

result = WAVELET(*h*, *waveletstruct*, /*Backwards*)

Input Parameters

h — A perfect reconstruction quadrature mirror filter in a filter structure, such as may be obtained using QMFDESIGN.

Parameters specific to the forward wavelet transform (the default transform):

> ***x*** — The data sequence to be transformed.
>
> ***n*** — (scalar) The number wavelet transform levels.

Parameters specific to the backward wavelet transform (with the *Backwards* keyword):

> ***waveletstruct*** — The output of the forward transform.

NOTE For the backward transform, the input parameter *h* must be a digital filter structure designed using QMFDESIGN.

Returned Value

Forward wavelet transform (default):

result — A data structure containing the coefficients of the forward wavelet transform.

Backward wavelet transform (with the *Backwards* keyword):

result — (array) The signal reconstructed from the wavelet transform coefficients.

Keywords

Backwards — If present and nonzero, the original signal is reconstructed from the wavelet transform coefficients.

Discussion

Computing the wavelet transform of a signal using a compactly supported orthornomal wavelet is equivalent to applying the quadrature mirror filter-bank structure shown in *Figure 2-42* to the signal.

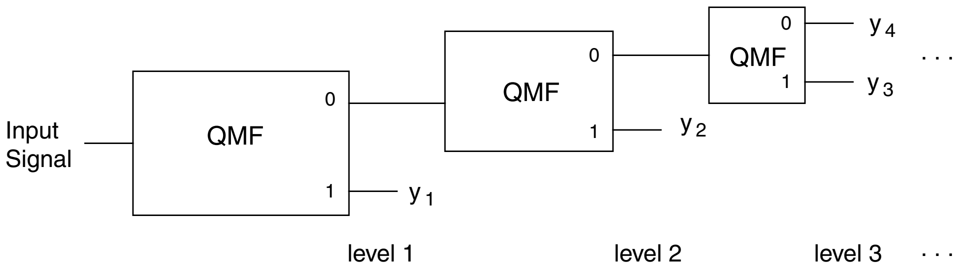


Figure 2-42 Filter structure for computing the forward wavelet transform.

The details of how and why the structure in *Figure 2-42* is connected to a compactly supported orthonomal wavelet are found in Akansu and Haddad, 1992; Daubechies, 1988, and 1992; Rioul and Vetterli, 1991; and Vaidyanathan, 1993.

The input and output sequences of each block in *Figure 2-42* represents the inputs and output of the forward part of the QMF structure discussed in the procedure QMF, *Figure 2-32* on page 178. The specific input-output relation between each block and the filter structure shown in *Figure 2-32* is illustrated in *Figure 2-43*.

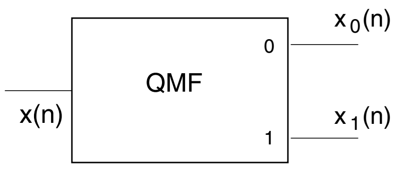


Figure 2-43 Basic computational cell in the forward wavelet transform structure.

WAVELET requires a quadrature mirror filter be supplied. Such a filter is obtained by using the QMFDESIGN function.

The output sequences $x_1, x_2, x_3, \ldots$ are the coefficients of the wavelet transform. The input parameter *n* specifies the number of levels of the wavelet transform structure to compute. For *n* levels of the wavelet transform, *n* + 1 output sequences are generated. WAVELET returns a PV-WAVE data structure. PARSEWAVELET provides access to the information in that data structure.

The backwards wavelet transform is computed using the filter bank structure shown in *Figure 2-44*.

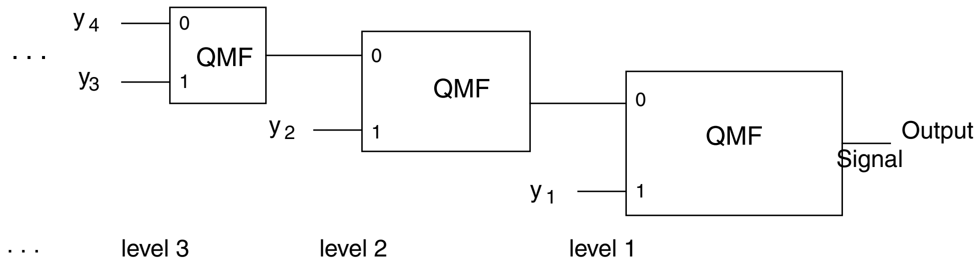


Figure 2-44 Filter structure for computing the backward wavelet transform.

The input and output sequences of each block in *Figure 2-44* represents the inputs and output of the backward part of the QMF structure discussed in the procedure QMF, *Figure 2-32 on page 178*. The specific input-output relation between each block and the filter structure in *Figure 2-32* is illustrated in *Figure 2-45*.

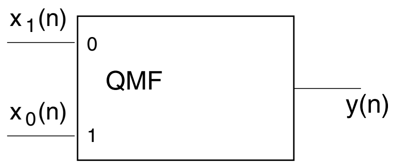


Figure 2-45 Basic computational element in the backward wavelet transform structure.

Example

This example illustrates how to compute the wavelet transform of a speech segment.

(UNIX)
```
file = '$VNI_DIR/sigpro-1_1/test/voice.dat'
```

(OpenVMS)
```
file = 'VNI_DIR:[SIGPRO-1_1.TEST]VOICE.DAT'
```

(Windows)
```
file = GETENV('VNI_DIR')+ $
'\sigpro-1_1\test\voice.dat'
```

```
OPENR, 1, file

x = BYTARR(7519)

READU, 1, x

CLOSE, 1

x = DOUBLE(x)
```
; Read in voice signal saying “PV-WAVE.”

```
m = 5

kz = [1.d, 1.d]

FOR i = 1, m-1 DO Kz = P_MULT([1.d, 1.d], kz)
```
; Determine the coefficients of the polynomial $K(z) = (1 + z^{-1})^m$
; to design the Daubechies QMF that generates wavelet
; with *m* vanishing moments.

```
h = QMFDESIGN(kz)
```
; Design a QMF using the factor $K(z)$.

```
n = 4

wstruct = WAVELET(h, x, n)
```
; Compute a 4-level wavelet transform.

```
x1 = PARSEWAVELET(wstruct, 1)

x2 = PARSEWAVELET(wstruct, 2)

x3 = PARSEWAVELET(wstruct, 3)

x4 = PARSEWAVELET(wstruct, 4)

x5 = PARSEWAVELET(wstruct, 5)
```
; Extract the different coefficient sequences from the wavelet
; data structure.

```
!P.Multi = [0, 1, 3]

PLOT, x, XStyle = 1, Title = 'Original Signal'

PLOT, x2, XStyle = 1, Title = $
   'Wavelet Coefficients, Level 2'

PLOT, x4, XStyle = 1, Title = $
   'Wavelet Coefficients, Level 4',$
   XTitle = 'Time'
```
; Plot the original signal (*Figure 2-46* (a)) and the wavelet
; coefficients from levels 2 and 4 (*Figure 2-46* (b) and (c),
; respectively).

```
y = WAVELET(h, wstruct, /Backward)
```
; Compute the inverse wavelet transform.

```
PM, MAX(ABS(x-y)), Title = $
   'Maximum Reconstruction Error'
```

```
Maximum Reconstruction Error
    5.4001248e-13
    ; Compute the maximum error in computing the forward
    ; and backward wavelet transform.
```

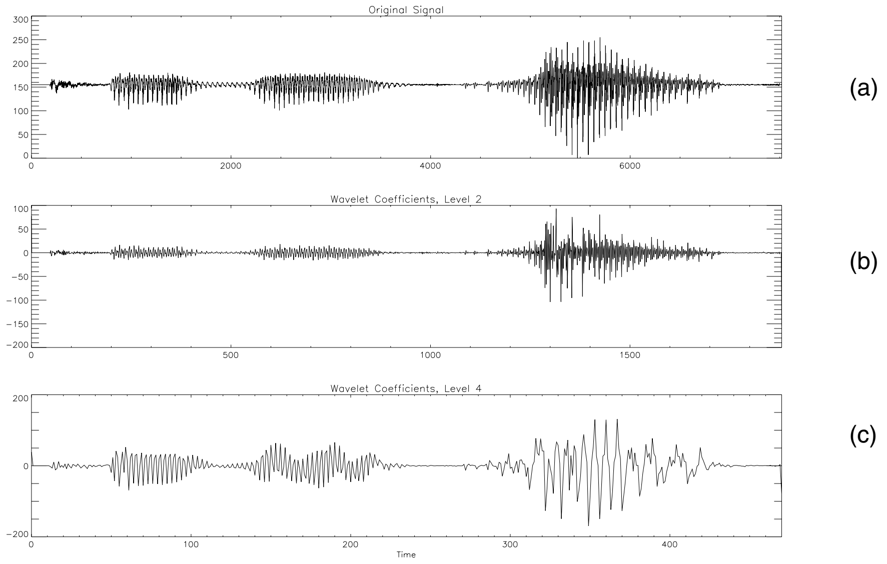


Figure 2-46 (a) The plot of the original voice signal saying “PV-WAVE,” and (b) its level 2 and (c) level 4 wavelet coefficients.

See Also

FILTDOWN, FILTUP, PARSEWAVELET, QMF, QMFDESIGN

For Additional Information

Akansu and Haddad, 1992.

Daubechies, 1988, and 1992.

Rioul and Vetterli, 1991.

Vaidyanathan, 1993.

ZEROPOLY Function

Finds the zeros of a polynomial with real or complex coefficients using the companion matrix method or, optionally, the Jenkins-Traub, three-stage algorithm.

Usage

result = ZEROPOLY(*coef*)

Input Parameters

coef — An array containing the coefficients of the polynomial in increasing order by degree. The polynomial is of the form

$$coef\,(n)\,z^n + coef\,(n-1)\,z^{(n-1)} + \ldots + coef\,(0)\ .$$

Returned Value

result —The complex array of zeros of the polynomial.

Keywords

Companion — If present and nonzero, the companion matrix method is used. (Default: companion matrix method)

Double — If present and nonzero, double precision is used.

Jenkins_Traub — If present and nonzero, the Jenkins-Traub, three-stage algorithm is used.

Discussion

ZEROPOLY computes the n zeros of the polynomial

$$p\,(z) = a_n z^n + a_{n-1} z^{(n-1)} + \ldots + a_1 z + a_0$$

where the coefficients a_i for $i = 0, 1, \ldots, n$ are real and n is the degree of the polynomial.

The default method used by ZEROPOLY is the companion matrix method. The companion matrix method is based on the fact that if C_a denotes the companion matrix associated with $p(z)$, then
det $(zI - C_a) = p(z)$, where I is an n-by-n identity matrix. Thus,
det $(z_0 I - C_a) = 0$ if, and only if, z_0 is a zero of $p(z)$. This implies that computing the eigenvalues of C_a will yield the zeros of $p(z)$. This method is thought to be more robust than the Jenkins-Traub algorithm in most cases, but the companion matrix

method is not as computationally efficient. Thus, if speed is a concern, the Jenkins-Traub algorithm should be considered.

If the keyword *Jenkins_Traub* is set, ZEROPOLY uses the Jenkins-Traub three-stage algorithm (Jenkins and Traub, 1970; Jenkins, 1975). The zeros are computed one-at-a-time for real zeros or two-at-a-time for a complex conjugate pair. As the zeros are found, the real zero or quadratic factor is removed by polynomial deflation.

Warning Errors

`MATH_ZERO_COEFF` — The first several coefficients of the polynomial are equal to zero. Several of the last roots are set to machine infinity to compensate for this problem.

`MATH_FEWER_ZEROS_FOUND` — Fewer than (N_ELEMENTS (*coef*) – 1) zeros were found. The root array contains the value for machine infinity in the locations that do not contain zeros.

Example

This example finds the zeros of the third-degree polynomial

$$p(z) = z^3 - 3z^2 + 4z - 2$$

where z is a complex variable.

```
coef = [-2, 4, -3, 1]
    ; Set the coefficients.
zeros = ZEROPOLY(coef)
    ; Compute the zeros.
PM, zeros, Title = 'The complex zeros found are: '
    The complex zeros found are:
    (      1.00000,      0.00000)
    (      1.00000,     -1.00000)
    (      1.00000,      1.00000)
        ; Print the complex polynomial results.
```

See Also

ROOT2POLY

For Additional Information

Jenkins, 1975.

Jenkins and Traub, 1970.

Bibliography

Ahrens, J. H., and Dieter, U. (1974), Computer methods for sampling from gamma, beta, Poisson, and binomial distributions, *Computing*, **12**, 223–246.

Akansu, A.N, and Haddad, R. A. (1992), *Multiresolution Signal Decomposition: Transforms, Subbands, and Wavelets*, Academic, Boston, MA.

Anderson, T.W. (1971), *The Statistical Analysis of Time Series*, John Wiley & Sons, New York, NY.

Atkinson, A.C. (1979), A family of switching algorithms for the computer generation of beta random variates, *Biometrika*, **66**, 141–145.

Blahut, R. E. (1985), “Fast Algorithms for Digital Signal Processing”, Addison-Wesley, Reading, MA.

Box, George E.P., and Gwilyn M. Jenkins (1976), *Time Series Analysis: Forecasting and Control*, revised ed., Holden-Day, Oakland, CA.

Cheng, R.C.H. (1978), Generating beta variates with nonintegral shape parameters, *Communications of the ACM*, **21**, 317–322.

Chui, C. K., (1992) *An Introduction to Wavelets*, Academic Press, New York, NY.

Constantinides, A. G. (1970), Spectral transformations for digital filters, *Proceedings of the IEEE*, 117, **8**, 1585–1590.

Cooley, J.W., and J.W. Tukey (1965), An algorithm for the machine computation of complex Fourier series, *Mathematics of Computation*, **19**, 297–301.

Crochiere, R. E., and L. R. Rabiner (1975), Optimum FIR Digital Filter Implementations for Decimation, Interpolation, and Narrow-Band Filtering, *IEEE Transactions on Acoustics, Speech, and Signal Processing,* ASSP–23, **5**, 444–456.

Crochiere, R. E., and L. R. Rabiner (1976), Further Considerations in the Design of Decimators and Interpolators, *IEEE Transactions on Acoustics, Speech, and Signal Processing*, ASSP–24, **4**, 296–311.

Crochiere, R. E., and L. R. Rabiner (1981), Interpolation and Decimation of Digital Signals—A Tutorial Review, *Proceedings of the IEEE*, 69, **3**, 300–331.

Daubechies, I. (1988), Orthonormal Bases of Compactly Supported Wavelets, *Communications on Pure Applied Mathematics,* **41**, 909–996.

Daubechies, I. (1992), Ten Lectures on Wavelets, Society for Industrial and Applied Mathematics, Philadelphia, PA.

de Boor, Carl (1978), *A Practical Guide to Splines*, Springer-Verlag, Berlin, Germany.

Demeure, C. J., and Mullis, C. T. (1990), A Newton-Raphson Method for Moving-Average Spectral Factorization Using the Euclid Algorithm, *IEEE Transactions on Acoustics, Speech, and Signal Processing,* 38, **10**, 1697–1709.

Dennis, J.E., Jr., and Robert B. Schnabel (1983), *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*, Prentice-Hall, Englewood Cliffs, NJ.

Digital Signal Processing Committee (1979), "Programs for Digital Signal Processing", IEEE Press, New York, NY.

Duell, K. A. (1994), Statistical Techniques for Processing Non-stationary Signals: Wavelets and Decision Theory, Ph.D. Thesis, Department of Electrical and Computer Engineering, University of Colorado at Boulder, Boulder, CO.

Franchitti, J. C. (1985), All-Pass Filter Interpolation and Frequency Transformation Problems, M.S. Thesis, Department of Electrical Engineering, University of Colorado at Boulder, Boulder, CO.

Gabel, R. A., and R. A. Roberts (1987), *Signals and Linear Systems*, Third Edition., John Wiley & Sons, New York, NY.

Gill, P.E., W. Murray, M.A. Saunders, and M.H. Wright (1985), Model building and practical aspects of nonlinear programming, *Computational Mathematical Programming*, (edited by K. Schittkowski), NATO ASI Series, **15**, Springer-Verlag, Berlin, Germany.

Goldfarb, D., and A. Idnani (1983), A numerically stable dual method for solving strictly convex quadratic programs, *Mathematical Programming*, **27**, 1–33.

Hamming, R. W. (1989), *Digital Filters*, Third Edition, Prentice-Hall, Englewood Cliffs, NJ.

Harris, F. J. (1978), On the Use of Windows for Harmonic Analysis with the Discrete Fourier Transform, *Proceedings of the IEEE*, 66, **1**, 51–83

Hildebrand, F.B. (1956), *Introduction to Numerical Analysis*, McGraw-Hill, New York, NY.

Jackson, L. B. (1991), *Signals, Systems, and Transforms*, Addison-Wesley, Reading, MA.

Jayant, N. S., and P. Noll (1984), *Digital Coding of Waveforms: Principles and Applications to Speech and Video*, Prentice-Hall, Englewood Cliffs, NJ.

Jenkins, M.A. (1975), Algorithm 493: Zeros of a real polynomial, *ACM Transactions on Mathematical Software*, **1**, 178–189.

Jenkins, M.A., and J.F. Traub (1970), A three-stage algorithm for real polynomials using quadratic iteration, *SIAM Journal on Numerical Analysis*, **7**, 545–566.

Jöhnk, M.D. (1964), Erzeugung von Betaverteilten und Gammaverteilten Zufalls-zahlen, *Metrika*, **8**, 5–15.

Kay, S. M. (1993), *Fundamentals of Statistical Signal Processing: Estimation Theory*, Prentice-Hall, Englewood Cliffs, NJ.

Kay, S. M. (1987), *Modern Spectral Estimation: Theory and Application*, Prentice-Hall, Englewood Cliffs, NJ.

Levenberg, K. (1944), A method for the solution of certain problems in least squares, *Quarterly of Applied Mathematics*, **2**, 164–168.

Lewis, P.A.W., A.S. Goodman, and J.M. Miller (1969), A pseudorandom number generator for the System/360, *IBM Systems Journal*, **8**, 136–146.

Luenberger, D. G. (1984), *Linear and Nonlinear Programming*, Second Edition, Addison-Wesley, Reading, MA.

Marple, S. L. (1987), *Digital Spectral Analysis with Applications*, Prentice-Hall, Englewood Cliffs, NJ.

Marquardt, D. (1963), An algorithm for least-squares estimation of nonlinear parameters, *SIAM Journal on Applied Mathematics*, **11**, 431–441.

McClellan, J. H., T. W. Parks, L. R. Rabiner (1973), A Computer Program for Designing Optimum FIR Linear Phase Digital Filters, *IEEE Transactions on Audio and Electroacoustics*, AU–21, **6**.

McClellan, J. H., T. W. Parks, L. R. Rabiner (1979), FIR linear phase filter design program. In *Programs for Digital Signal Processing*, ed. Digital Signal Processing Committee, IEEE Acoustics, Speech, and Signal Processing Society. IEEE Press, New York, NY.

Mitra, S. K., and J. F. Kaiser (1993), *Handbook for Digital Signal Processing*, John Wiley & Sons, New York, NY.

Moré, Jorge, Burton Garbow, and Kenneth Hillstrom (1980), *User Guide for MINPACK-1*, Argonne National Laboratory Report ANL 80–74, Argonne, IL.

Murtagh, Bruce A. (1981), *Advanced Linear Programming: Computation and Practice*, McGraw-Hill, New York, NY.

Murty, Katta G. (1983), *Linear Programming*, John Wiley & Sons, New York, NY.

Nawab, S. H., and T. F. Quatieri (1988), Short-time Fourier Transform, (edited by J. S. Lim and A. V. Oppenheim), *Advanced Topics in Signal Processing*, Prentice-Hall, Englewood Cliffs, NJ.

Oetken, G., Parks, T. W., and Schussler, H. S. (1975), New results in the design of digital interpolators, *IEEE Transactions on Acoustics, Speech and Signal Processing*, ASSP-23, **3**.

Oppenheim, A. V., and Schafer, R, W. (1975), *Digital Signal Processing*, Prentice-Hall, Englewood Cliffs, NJ.

Oppenheim, A. V., and Schafer, R. W. (1989), *Discrete-Time Signal Processing*, Prentice-Hall, Englewood Cliffs, NJ.

Oppenheim, A. V., and Willsky, A. S., and Young, I. T. (1983), *Signal and Systems*, Prentice-Hall, Englewood Cliffs, NJ.

Parks, T. W., and C.S. Burrus (1987), *Digital Filter Design*, John Wiley & Sons, New York, NY.

Piessens, R., E. deDoncker-Kapenga, C.W. Überhuber, and D.K. Kahaner (1983), *QUADPACK*, Springer-Verlag, Berlin, Germany.

Porat, B. (1994), *Digital Processing of Random Signals: Theory and Methods*, Prentice-Hall, Englewood Cliffs, NJ.

Powell, M.J.D. (1978), A fast algorithm for nonlinearly constrained optimization calculations, in *Numerical Analysis Proceedings, Dundee 1977, Lecture Notes in Mathematics*, (edited by G. A. Watson), **630**, 144–157, Springer-Verlag, Berlin, Germany.

Powell, M.J.D. (1983), *ZQPCVX a FORTRAN subroutine for convex quadratic programming*, DAMTP Report 1983/NA17, University of Cambridge, Cambridge, England.

Powell, M.J.D. (1985), On the quadratic programming algorithm of Goldfarb and Idnani, *Mathematical Programming Study*, **25**, 46–61.

Proakis, J. G., and D. G. Manolakis (1992), *Digital Signal Processing: Principles, Algorithms, and Applications*, Second Edition, MacMillan, New York, NY.

Programs for Digital Signal Processing (1979), IEEE Press, John Wiley & Sons, New York, NY.

Rabiner, L. R., and B. Gold (1975), *Theory and Application of Digital Signal Processing*, Prentice-Hall, Englewood Cliffs, NJ.

Rioul, O, and Vetterli, M., (1991) Wavelets and Signal Processing, *IEEE Signal Processing Magazine*, October, 14–38.

Roberts, R. A., C. T. Mullis (1987), *Digital Signal Processing*. Addison-Wesley, Reading, MA.

Savitzky, A. and Golay, M. J. E (1964), Smoothing and Differentiation of Data by Simplified Least Squares Procedures, *Analytical Chemistry*, 36, **8**, 1627–1639.

Scharf, L. L. (1991), *Statistical Signal Processing: Detection, Estimation, and Time Series Analysis*, Addison-Wesley, Reading, MA.

Schittkowski, K. (1983), On the convergence of a sequential quadratic programming method with an augmented Lagrangian line search function, *Mathematik Operations for Schung and Statistik, Serie Optimization*, **14**, 197–216.

Schittkowski, K. (1986), NLPQL: A FORTRAN subroutine solving constrained nonlinear programming problems, (edited by Clyde L. Monma), *Annals of Operations Research*, **5**, 485–500.

Schmeiser, Bruce (1983), Recent advances in generating observations from discrete random variates, *Computer Science and Statistics: Proceedings of the Fifteenth Symposium on the Interface*, (edited by James E. Gentle), North-Holland Publishing Company, Amsterdam, 154–160.

Schmeiser, Bruce W., and A.J.G. Babu (1980), Beta variate generation via exponential majorizing functions, *Operations Research*, **28**, 917–926.

Schmeiser, Bruce W., and Ram Lal (1980), Squeeze methods for generating gamma variates, *Journal of the American Statistical Association*, **75**, 679–682.

Schmeiser, Bruce, and Voratas Kachitvichyanukul (1981), *Poisson Random Variate Generation*, Research Memorandum 81-4, School of Industrial Engineering, Purdue University, West Lafayette, IN.

Schussler, H. W., and P. Steffen (1988), Some Advanced Topics in Filter Design, (edited by J. S. Lim and A. V. Oppenheim), *Advanced Topics in Signal Processing*, Prentice-Hall, Englewood Cliffs, NJ.

Steinier, J., and Y. Termonia and J. Deltour (1972), Comments on Smoothing and Differentiation of Data by Simplified Least Squares Procedures, *Analytical Chemistry*, 44, **11**, 1906–1909.

Stoer, J. (1985), Principles of sequential quadratic programming methods for solving nonlinear programs, *Computational Mathematical Programming*, (edited by K. Schittkowski), NATO ASI Series, **15**, Springer-Verlag, Berlin, Germany.

Therrien, C. W. (1992), *Discrete Random Signals and Statistical Signal Processing*, Prentice-Hall, Englewood Cliffs, NJ.

Vaidyanathan, P. P. (1993), *Multirate Systems and Filter Banks*, Prentice-Hall, Englewood Cliffs, NJ.

Welch, P. D. (1967), The Use of Fast Fourier Transform for the Estimation of Power Spectra: A Method Based on Time Averaging Over Short, Modified Periodograms, *IEEE Transactions on Audio Electroacoustics*, AU-15, **2**, 70–73.

Related Routines

This appendix lists routines from PV-WAVE, PV-WAVE:IMSL Statistics, and PV-WAVE:IMSL Statistics that can be useful in digital signal processing applications. For detailed information on these routines, refer to the *PV-WAVE Reference, PV-WAVE:IMSL Statistics Reference,* and the *PV-WAVE:IMSL Statistics Reference*.

PV-WAVE Routines

ABS(x)

Evaluates the absolute value function.

ATAN(x [, y])

Returns the angle, whose tangent is x, expressed in radians. If two parameters are supplied, the angle whose tangent is equal to *y/s* is returned. The range of ATAN is between $-\pi/2$ and $\pi/2$ for the single argument case and between $-\pi$ and π if two arguments are given.

COMPLEX(real [, imaginary])

Converts an expression to complex data type.

DERIV([x,]y)

Performs numerical differentiation using three-point Lagrangian interpolation.

FFT(array, direction)

Returns the Fast Fourier Transform for the input variable.

HANNING(col [, row])

Standard Library function that implements a window function for Fast Fourier Transform signal or image filtering.

POLY(x, coefficients)

Standard Library function that evaluates a polynomial function of a variable.

REVERSE(array, dimension)

Standard Library function that reverses a vector or array for a given dimension.

PV-WAVE:IMSL Statistics Routines

ARMA(array, p, q)

Computes method-of-moments or least-squares estimates of parameters for a nonseasonal ARMA model.

CHISQCDF(expr, n)

Evaluates the chi-squared distribution function. Using a keyword, the inverse of the chi-squared distribution can be evaluated.

PV-WAVE:IMSL Mathematics Routines

BESSI(order, z)

Evaluates a modified Bessel function of the first kind with real order and real or complex parameters.

CHISQCDF(expr, n)

Evaluates the chi-squared distribution function. Using a keyword, the inverse of the chi-squared distribution can be evaluated.

CHNNDSOL(matrix [, matrix])

Solves a real symmetric nonnegative definite system of linear equations Ax = b. Computes the solution to Ax = b given the Cholesky factor.

CSINTERP(array, array)

Computes a cubic spline interpolant, specifying various endpoint conditions. The default interpolant satisfies the not-a-knot condition.

EIG(matrix)

Computes the eigenexpansion of a real or complex matrix A. If the matrix is known to be symmetric or Hermitian, a keyword can be used to trigger more efficient algorithms.

EIGSYMGEN(matrix, matrix)

Computes the generalized eigenexpansion of a system $Ax = \lambda Bx$. The matrices A and B are real and symmetric, and B is positive definite.

ERF(x)

Evaluates the real error function $\mathrm{erf}(x)$. Using a keyword, the inverse error function $\mathrm{erf}^{-1}(x)$ can be evaluated.

ERFC(x)

Evaluates the real complementary error function erfc(x). Using a keyword, the inverse complementary error function erfc^{-1}(x) can be evaluated.

GAMMA(x)

Evaluates the real gamma function $\Gamma(x)$.

GAMMACDF(x, +param)

Evaluates the gamma distribution function.

LNGAMMA(x)

Evaluates the logarithm of the absolute value of the gamma function log$|\Gamma(x)|$.

NORMALCDF(expr)

Evaluates the standard normal (Gaussian) distribution function. Using a keyword, the inverse of the standard normal (Gaussian) distribution can be evaluated.

SPINTEG(a, b, c, d, structure)

Computes the integral of a one- or two-dimensional spline. (PV-WAVE Reference)

SVDCOMP(matrix)

Computes the singular value decomposition (SVD), $A = USV^T$, of a real or complex rectangular matrix A. An estimate of the rank of A also can be computed.

Signal Processing Index